

# **Distributed System for Factorisation of Large Numbers**

A Master's Thesis Performed at the  
Division of Information Theory by

Angela Johansson

LiTH-ISY-EX-3505-2004  
Linköping, 2004



# **Distributed System for Factorisation of Large Numbers**

A Master's Thesis Performed at the  
Division of Information Theory by


Angela Johansson

LiTH-ISY-EX-3505-2004

Supervisor: Viiveke Fåk  
Examiner: Viiveke Fåk

Linköping, 28 May, 2004



 <b>LINKÖPINGS UNIVERSITET</b>	<b>Avdelning, Institution</b> Division, Department  Institutionen för systemteknik 581 83 LINKÖPING	<b>Datum</b> Date 2004-05-28						
<b>Språk</b> Language Svenska/Swedish X Engelska/English	<b>Rapporttyp</b> Report category Licentiatavhandling X Examensarbete C-uppsats D-uppsats Övrig rapport _____	<table border="1"> <tr> <td colspan="2"><b>ISBN</b></td> </tr> <tr> <td colspan="2"><b>ISRN</b> LITH-ISY-EX-3505-2004</td> </tr> <tr> <td><b>Serietitel och serienummer</b> Title of series, numbering</td> <td><b>ISSN</b> _____</td> </tr> </table>	<b>ISBN</b>		<b>ISRN</b> LITH-ISY-EX-3505-2004		<b>Serietitel och serienummer</b> Title of series, numbering	<b>ISSN</b> _____
<b>ISBN</b>								
<b>ISRN</b> LITH-ISY-EX-3505-2004								
<b>Serietitel och serienummer</b> Title of series, numbering	<b>ISSN</b> _____							
<b>URL för elektronisk version</b> <a href="http://www.ep.liu.se/exjobb/isy/2004/3505/">http://www.ep.liu.se/exjobb/isy/2004/3505/</a>								
<table border="1"> <tr> <td><b>Titel</b> Title</td> <td>Distributed System for Factorisation of Large Numbers</td> </tr> <tr> <td><b>Författare</b> Author</td> <td>Angela Johansson</td> </tr> </table>			<b>Titel</b> Title	Distributed System for Factorisation of Large Numbers	<b>Författare</b> Author	Angela Johansson		
<b>Titel</b> Title	Distributed System for Factorisation of Large Numbers							
<b>Författare</b> Author	Angela Johansson							
<b>Sammanfattning</b> Abstract <p>This thesis aims at implementing methods for factorisation of large numbers. Seeing that there is no deterministic algorithm for finding the prime factors of a given number, the task proves rather difficult. Luckily, there have been developed some effective probabilistic methods since the invention of the computer so that it is now possible to factor numbers having about 200 decimal digits. This however consumes a large amount of resources and therefore, virtually all new factorisations are achieved using the combined power of many computers in a distributed system.</p> <p>The nature of the distributed system can vary. The original goal of the thesis was to develop a client/server system that allows clients to carry out a portion of the overall computations and submit the result to the server.</p> <p>Methods for factorisation discussed for implementation in the thesis are: the quadratic sieve, the number field sieve and the elliptic curve method. Actually implemented was only a variant of the quadratic sieve: the multiple polynomial quadratic sieve (MPQS).</p>								
<b>Nyckelord</b> Keyword factorisation, factorization, prime factor, quadratic sieve, QS, MPQS, number field sieve, elliptic curve method								



## Acknowledgements

---

I want to thank my husband for supporting me and helping me debug the program. Without him, I would have given up at the end of the first week of implementation work.

Thanks also to my examiner/supervisor Viiveke Fåk for giving me the opportunity to do this thesis work and to all the people who answered my questions the best they could, in person or by email. A special thanks to Jacob Löfvenberg, Danyo Danev, Peter Hackman, Damian Weber and Scott Contini.

I am thankful to the National Supercomputer Centre in Sweden (NSC) for providing an account and computing resources on the Linux cluster *Monolith* and the SGI3800. That is how I could gather so many results from test runs for my results chapter.

My acknowledgements to the ones that wrote software this system is based upon, like the LIP by Arjen K. Lenstra.

And last, but not least, I am grateful to my mother and stepfather for making it possible for me to come to Sweden at all. Thank you for financing my studies and helping me make my dreams come true.





## Table of Contents

---

<b>1 Introduction .....</b>	<b>1</b>
1.1 Task .....	1
1.2 Brief History .....	2
1.3 Existing Systems .....	5
1.4 Document Outline .....	7
1.5 Glossary .....	7
<b>2 The Quadratic Sieve .....</b>	<b>11</b>
2.1 The Method .....	11
2.2 Implementation.....	13
<b>3 Implementation Details .....</b>	<b>19</b>
3.1 LIP .....	19
3.2 LiDIA .....	19
3.3 Other Software .....	20
3.4 Environment.....	21
<b>4 Design .....</b>	<b>25</b>
4.1 Code Structure .....	25
4.2 Data Structure .....	29
4.3 Network protocol .....	30
<b>5 Results.....</b>	<b>37</b>
5.1 General Observations.....	37
5.2 Block Size Comparison .....	47
5.3 Sieving Bound Comparison .....	52
5.4 Large Prime Bound Comparison.....	60
5.5 Factor Base Size Comparison.....	67
5.6 Environment Comparison.....	73
<b>6 Conclusions .....</b>	<b>77</b>
6.1 Results .....	77
6.2 Personal Experiences.....	78
<b>7 Future Work .....</b>	<b>79</b>

<b>8 References .....</b>	<b>81</b>
8.1 Books .....	81
8.2 Internet .....	81
8.3 Publications .....	82
<b>Appendix A - Early Factorisation Methods .....</b>	<b>85</b>
<b>Appendix B - Modern Factorisation Methods .....</b>	<b>87</b>
<b>Appendix C - Definitions .....</b>	<b>89</b>
<b>Appendix D - The Number Field Sieve .....</b>	<b>99</b>
<b>Appendix E - The Elliptic Curve Method .....</b>	<b>105</b>

## List of Definitions

---

1	Factor Base .....	11
2	Quadratic Residue .....	89
3	Legendre's Symbol .....	89
4	Jacobi's Symbol .....	89
5	Continued Fraction.....	90
6	Partial Numerator/Denominator.....	91
7	Regular Continued Fraction.....	91
8	Regular Continued Fraction Expansion .....	91
9	Elliptic Curve.....	92
10	Addition of Points.....	93
11	Infinity Point.....	93
12	Multiple of a Point .....	93
13	Homogeneous Coordinates.....	93
14	Quadratic Field.....	94
15	Integer of the Quadratic Field.....	94
16	Conjugate in the Quadratic Field .....	94
17	Norm in the Quadratic Field.....	94
18	Unit of the Quadratic Field.....	95
19	Associated Integers in the Quadratic Field.....	95
20	Prime/Composite in the Quadratic Field .....	95
21	Algebraic Number .....	95
22	Conjugate of an Algebraic Number .....	95
23	Algebraic Number Field .....	95
24	Algebraic Integer .....	95
25	Ring of Algebraic Integers.....	96
26	Norm in the Number Field.....	96
27	B-smooth .....	96
28	Ideal .....	96
29	Norm of an Ideal.....	96
30	First Degree Prime Ideal .....	96
31	Ring Homomorphism .....	99



## List of Methods

---

Continued Fraction Algorithm (CFRAC) .....	88
Elliptic Curve Method (ECM) .....	105
Fermat's Method .....	85
Gauss' Method .....	86
General Number Field Sieve (GNFS) .....	100
Legendre's Method .....	86
Multiple Polynomial Quadratic Sieve (MPQS) .....	12
Number Field Sieve (NFS) .....	99
Pollard p-1 .....	87
Pollard Rho .....	87
Quadratic Sieve (QS) .....	11
Trial Division .....	85



## List of Tables

---

1	Available Compilation/Runtime Environment.....	21
2	nsieve Benchmarking Results. ....	23
3	Messages Included in the Network Protocol.....	31
4	Results of the First Number Size Comparison Test Runs.....	37
5	Results of the Second/Third Number Size Comparison Test Runs.....	39
6	Results of the Fourth/Fifth Number Size Comparison Test Runs.....	40
7	Results of the Sixth/Seventh Number Size Comparison Test Runs.....	42
8	Optimal Sieving Bounds.....	44
9	Results of the Eighth Number Size Comparison Test Runs.....	44
10	Results of the Ninth Number Size Comparison Test Runs.....	45
11	Parameters of the First Block Size Comparison Test Runs.....	47
12	Results of the First Block Size Comparison Test Runs.....	47
13	Results of the Second Block Size Comparison Test Runs.....	50
14	Parameters of the First Sieving Bound Comparison Test Runs.....	52
15	Results of the First Sieving Bound Comparison Test Runs.....	52
16	Parameters of the Second Sieving Bound Comparison Test Runs.....	56
17	Results of the Second Sieving Bound Comparison Test Runs.....	56
18	Parameters of the Third Sieving Bound Comparison Test Runs.....	58
19	Results of the Third Sieving Bound Comparison Test Runs.....	58

20	Parameters of the First Large Prime Bound Comparison Test Runs.....	60
21	Results of the First Large Prime Bound Comparison Test Runs.....	60
22	Parameters of the Second Large Prime Bound Comparison Test Runs.....	63
23	Results of the Second Large Prime Bound Comparison Test Runs.....	63
24	Parameters of the Third Large Prime Bound Comparison Test Runs.....	65
25	Results of the Third Large Prime Bound Comparison Test Runs.....	65
26	Parameters of the First Factor Base Size Comparison Test Runs.....	67
27	Results of the First Factor Base Size Comparison Test Runs.....	68
28	Parameters of the Second Factor Base Size Comparison Test Runs.....	70
29	Results of the Second Factor Base Size Comparison Test Runs.....	70
30	Some Optimal Parameters.....	72
31	Parameters of the System Comparison Test Runs. ....	73
32	Results of the System Comparison Test Runs. ....	73
33	Results of the Compiler Comparison Test Runs. ....	75



## List of Figures

---

1	UML Diagram of the Standalone Application. ....	26
2	UML Diagram of the Restructured Sieving Part. ....	27
3	UML Diagram of the Server Part. ....	28
4	ER-diagram for the database. ....	29
5	The Client's Flow Chart Diagram - Part 1. ....	35
6	The Client's Flow Chart Diagram - Part 2. ....	36
7	Diagram of the First Number Size Comparison Test Runs. ....	38
8	Diagram of the Second/Third Number Size Comparison Test Runs. ....	40
9	Diagram of the Fourth/Fifth Number Size Comparison Test Runs. ....	41
10	Diagram of the Sixth/Seventh Number Size Comparison Test Runs. ....	43
11	Diagram of the Eighth Number Size Comparison Test Runs. ....	45
12	Diagram of the Ninth Number Size Comparison Test Runs. ....	46
13	Diagram of the First Block Size Comparison Test Runs. ....	48
14	Diagram of the Second Block Size Comparison Test Runs. ....	51
15	Diagram of the First Sieving Bound Comparison Test Runs. ....	53
16	Pie Charts of the First Sieving Bound Comparison Test Runs. ....	54
16	Pie Charts of the First Sieving Bound Comparison Test Runs. ....	55
17	Diagram of the Second Sieving Bound Comparison Test Runs. ....	57
18	Diagram of the Third Sieving Bound Comparison Test Runs. ....	59
19	Diagram of the First Large Prime Bound Comparison Test Runs. ....	62
20	Diagram of the Second Large Prime Bound Comparison Test Runs. ....	64

21	Diagram of the Third Large Prime Bound Comparison Test Runs. ....	66
22	Diagram of the First Factor Base Size Comparison Test Runs. ....	69
23	Diagram of the Second Factor Base Size Comparison Test Runs. ....	71
24	Diagram of the System Comparison Test Runs.....	74
25	Diagram of the Compiler Comparison Test Runs.....	76

# 1 Introduction

---

This chapter contains a short description of the task, a brief history of the study of primes and factorisation, an overview of existing systems for factorisation, a document outline and a glossary.

It is not essential reading for understanding the rest of the thesis, but it outlines the base of the thesis and can therefore be useful.

## 1.1 Task

### 1.1.1 Background

The computer security algorithm RSA is widely used for public key cryptography and relies among other things on the difficulty of finding the prime factors of a given number. If there was a fast, deterministic way to calculate the prime factors, the algorithm would become totally useless. Instead, there have been some efforts to determine the factors with probabilistic methods conducting a “guided search” for candidate factors. Some of the methods implement fairly simple mathematical concepts, others (like the number field sieve) exploit the structure of complicated algebraic concepts.

### 1.1.2 Goal

This thesis with the title “Distributed System for Factorisation of Large Numbers” aims at the implementation of different methods for factorisation. The latest findings in research should be applied and the system should be able to send out portions of the search space to other computers (hence “distributed system”).

The task involves:

- Researching different methods for factorisation, such as ECM (elliptic curve method) and NFS (number field sieve).
- Choosing an adequate programming language and implementing the methods.
- Designing an application for running the program and receiving portions of the search space.
- Implementing a server that distributes the portions, keeps track of the progress and calculates the final result.

## 1.2 Brief History

Ground breaking work has been written as early as 300 B.C. by Euclid who studied the properties of the integers. He stated many theorems about primes and found an algorithm for calculating the greatest common divisor of two integers which is frequently used nowadays. Around 200 B.C., Eratosthenes constructed a method called the *Sieve of Eratosthenes* for finding all the primes up to a given number.

After that, it took a long time until research was resumed. The French monk Marin Mersenne wrote about primes of the form  $2^n - 1$  in 1644. Also in the early 17<sup>th</sup> century, Fermat wrote down a number of theorems about integers and primes and among other things, he developed a factorisation algorithm (see appendix A). In the 18<sup>th</sup> century, several contemporary mathematicians have contributed to the subject, for example Leonhard Euler, Adrien-Marie Legendre and Carl Friedrich Gauss. They each developed factorisation methods and carried out calculations by pen and paper. Research proceeded and methods got more sophisticated. Among other things, Edouard Lucas wrote down a theorem in 1870 which did not get published until Derrick Lehmer found a proof and wrote about the Lucas-Lehmer primality test in 1930.

*Sieving methods* began to evolve. The basics of the technique are due to Maurice Kraitchik (and, of course, Eratosthenes). From now on, algorithms for factoring large numbers were to be probabilistic instead of deterministic, such as J. M. Pollard's two factorisation methods written in the 1970s.

It was not until the invention of the computer that results became more accurate and numerous. Before that, there had been erroneous prime tables which led to miscalculations. Now, one could factor larger numbers with less effort and challenges grew steadily along with the growing computing power.

Research is ongoing and improvements to modern factorisation methods are developed constantly. Also, there are new insights and strategies in the choice of parameters.

### 1.2.1 Special Numbers

The ancient Greeks were very occupied with the beauty of things (and thus the beauty of numbers, too). That is why they searched for special relations between numbers. Here is an example:

A *perfect number* is a number which is the sum of all its divisors (i.e.  $6 = 1 + 2 + 3$  and  $6 = 1 \cdot 2 \cdot 3$ ). Today, we know that an even number is perfect iff it can be written as  $2^{n-1}(2^n - 1)$  with  $2^n - 1$  prime.

*Mersenne primes* are numbers  $2^n - 1$  that are prime. Mersenne, a French mathematician from the 16<sup>th</sup>/17<sup>th</sup> century, observed that those numbers (called *Mersenne numbers*) are prime for  $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127$  and 257 and composite for all other  $n < 257$ .

Numbers of the form  $F_n = 2^{2^n} + 1$  are called *Fermat numbers* and they are composite for  $5 \leq n \leq 23$ . Fermat himself believed that every *Fermat number* was prime. There is an interest in factoring these numbers, but currently only  $F_5$  to  $F_{11}$  have been completely factored.

The *Lucas sequence* consists of the numbers  $l_{n+1} = l_n + l_{n-1}$  with  $l_1 = 1$  and  $l_2 = 3$ . It can be used to test whether a given *Mersenne number* is prime.

In 1925, Allan Cunningham and H. J. Woodall published tables containing factorisations of  $b^n \pm 1$  for the bases  $b = 2, 3, 5, 6, 7, 10, 11, 12$  with various high powers of  $n$ . Such numbers are called *Cunningham numbers* and the *Cunningham project* aims at extending the tables further.

*RSA challenge numbers* are large numbers that are published by RSA laboratories and free for everybody to factor. There are even prizes for finding the factors. The numbers used to be labelled *RSA- $\langle$ number of decimal digits $\rangle$* , but right now they are labelled *RSA- $\langle$ number of binary digits $\rangle$* .

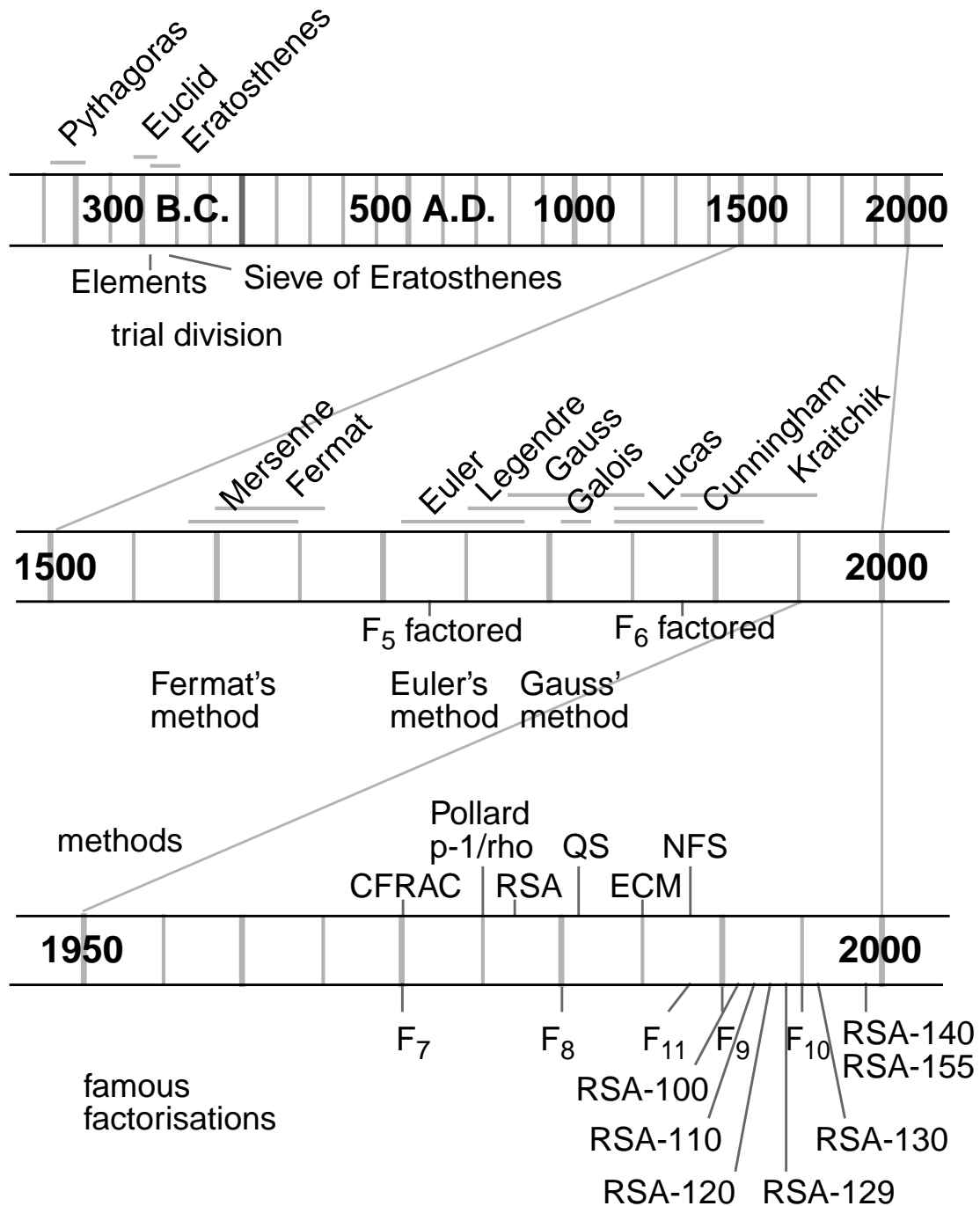
The current challenges are eight numbers between:

RSA-576 (174 decimal digits) - \$ 10.000 prize and

RSA-2048 (617 decimal digits) - \$ 200.000 prize

The reason why RSA laboratories wants people to factor their numbers is clearly that they want to be up to date when it comes to the progress in research and the computing power of modern machines. To ensure the security of the RSA algorithm, they need to adapt the key size from time to time. Interesting issues are: How large does a number have to get to be infeasible to factor? How much time does it take to factor a number of a certain size? Are there any numbers that are easier to factor than others?

## 1.2.2 Time Line



Some researchers of the 20<sup>th</sup> century:

Richard P. Brent	Arjen K. Lenstra	John M. Pollard
John Brillhart	Hendrik W. Lenstra Jr.	Carl Pomerance
Joe P. Buhler	Paul Leyland	R. E. Powers
Stefania Cavallar	Mark S. Manasse	Hans Riesel
Bruce A. Dodson	Peter L. Montgomery	Robert D. Silverman
Derrick H. Lehmer	Michael A. Morrison	Samuel S. Wagstaff

## 1.3 Existing Systems

### 1.3.1 NFSNET

From the NFSNET web page: “The goal of the NFSNET project is to use the Number Field Sieve to find the factors of increasingly large numbers.”

Anybody can participate in the current project and download client software (see [6]). At the time of writing, NFSNET works on factoring  $2^{811}-1$ , a number with 245 digits. The factorisation would establish a new worldwide SNFS record. The latest factorisation was achieved on December 2<sup>nd</sup>, 2003 when NFSNET completed the factorisation of  $2^{757}-1$ , a number with 213 digits.

Already in 1997, Sam Wagstaff could show up with a world record through NFSNET - the factorisation of  $(3^{349}-1)/2$ , a number with 167 decimal digits.

In 2002, NFSNET was revived and got its current form. The first factorisation performed then was that of the Woodall number  $W(668): 668 \cdot 2^{668}-1$ , a number with 204 decimal digits.

NFSNET describes its way of working like this: “Each factorization is defined by a “project” which holds information such as the number being factored, the polynomials used, the size of the factor bases and so forth. The servers’ responsibility is to provide project details to the clients, to allocate regions to be sieved, and to collect the results from the clients for further processing later.”

The distributed system for factorisation of large numbers described in this thesis should work in the same way, apart from the fact that it should be able to keep track of several projects at the same time. The intention is also that different projects can be carried through using different factorisation methods.

### 1.3.2 ECMNET

From the ECMNET web page: “Richard Brent has predicted in 1985 [...] that factors up to 50 digits could be found by the Elliptic Curve Method (ECM). [...] The original purpose of the ECMNET project was to make Richard’s prediction true, i.e. to find a factor of 50 digits or more by ECM. This goal was attained on September 14, 1998, when Conrad Curry found a 53-digit factor of  $2^{677}-1$  c150 using George Woltman’s mprime program. The new goal of ECMNET is now to find other large factors by ecm, mainly by contributing to the Cunningham project.”

ECMNET itself is thus not a centralised project like NFSNET but rather a repository for resources on the elliptic curve method and factorisation in general. Those who are interested can download GMP-ECM (see [7]) and run it on their computers to find factors independently from others. However, Tim Charron wrote a client which can be run against a master server run by Paul Leyland at Microsoft (see [8]). On the ECMNET web page, there are also links to other programs based on the elliptic curve method and a lot of information about special numbers can be found.

The current record for prime factors found via the elliptic curve method is a 54-digit factor of a 127-digit number, found in December 1999 by Nik Lygeros and Michel Mizony.

### 1.3.3 FermatSearch

As the name suggest, the FermatSearch project specialises on finding factors of *Fermat numbers*. It has been proven that all factors are of the form  $k2^n+1$ , so the FermatSearch program generates such numbers and looks for a factor by using modular arithmetic.

The project is not fully automated like NFSNET, but it is coordinated. After downloading the program (see [9]), the participants are asked to reserve a value range and later send the results back to the author of the page, Leonid Durman, via email.

### 1.3.4 The Cunningham Project

Unlike the previously mentioned projects, this project is passive instead of active. It does not contribute to factoring any numbers but is dedicated to bookkeeping.

From the project web page: “The Cunningham Project seeks to factor the numbers  $b^n \pm 1$  for  $b = 2, 3, 5, 6, 7, 10, 11, 12$ , up to high powers  $n$ . The Cunningham tables are the tables in the book “Factorizations of  $b^n \pm 1$ ,  $b = 2, 3, 5, 6, 7, 10, 11, 12$  up to high powers,”“ Sam Wagstaff currently maintains the tables and provides “most wanted” lists prepared by J. L. Selfridge (see [10]).

The Cunningham Project is described as “likely the longest, ongoing computational project in history”. As mentioned above, the project started in 1925 when Lt.-Col. Alan J.C. Cunningham and H. J. Woodall began writing down the first tables. The project grew popular over the years and today, most of the ongoing factorisation projects aim at filling the holes in the tables.



## 1.4 Document Outline

This section summarises the contents of the rest of this thesis.

Chapter 1, Introduction, gives information about the goal of this thesis and other useful information about this document. Furthermore, it contains some interesting things about factorisation in general.

Chapter 2, The Quadratic Sieve, contains a description of the quadratic sieve factorisation algorithm.

Chapter 3, Implementation Details, shows which software this system relies on and in which environments it will be tested.

Chapter 4, Design, gives an account of the actual implementation and sketches the system's features from a software engineering point of view.

Chapter 5, Results, deals with the readings from test runs and explains what happens when you change certain parameters.

Chapter 6, Conclusions, summarises the previous chapter and contains my personal experiences with this thesis.

Chapter 7, Future Work, is about ideas and hints for future work and discusses, according to the previous chapter, the parts of the thesis that were left out due to lack of time.

Chapter 8, References, contains literature references.

Appendix A, Early Factorisation Methods, gives an overview of some basic factorisation methods.

Appendix B, Modern Factorisation Methods, shows some modern factorisation methods.

Appendix C, Definitions, provides some theoretical background about quadratic residues, continued fractions, elliptic curves and number fields.

Appendix D, The Number Field Sieve, explains the number field sieve factorisation method.

Appendix E, The Elliptic Curve Method, gives an introduction to the elliptic curve method.

## 1.5 Glossary

### Continued Fraction

A *continued fraction* is a fraction of the form

$$b_0 + \frac{a_1}{b_1 + \frac{a_2}{b_2 + \dots + \frac{a_n}{b_n}}} \text{ and is written as } b_0 + \overline{b_1} + \overline{b_2} + \dots + \overline{b_n}.$$

See section C.2.

### Deterministic Algorithm

An algorithm is said to be *deterministic*, if it can give you a result in a specified amount of time for every input. This implies that it never runs forever or terminates without having found an appropriate result. By intuition, a *deterministic algorithm* can be called systematic.

### Elliptic Curve

*Elliptic curves* are curves represented by a cubic equation of the form:  $y^2 = Ax^3 + Bx^2 + Cx + D$ .

See section C.3.

### Elliptic Curve Method (ECM)

The *elliptic curve method* is a factorisation method that makes use of elliptic curves by calculating prime multiples of points on elliptic curves.

See appendix E.

### ER-Diagram

An *entity-relationship diagram* (called *ER-diagram*) is a way of representing the structure of a relational database. An entity is a discrete object and a relationship shows how two or more entities are related to one another.

### Factor Base

A *factor base* is a set of prime numbers that constitute a base for factorisation of relatively small function values in the quadratic sieve method and the number field sieve.

See section 2.1.

### Fermat Number

A number of the form  $F_n = 2^{2^n} + 1$  is called *nth Fermat number*.

**gcd**

The *greatest common divisor* (*gcd*) of two numbers is the largest integer that divides both of the two numbers. Ex.:  $\text{gcd}(10, 35) = 5$ .

**General Number Field Sieve (GNFS)**

The *general number field sieve* is a version of the *number field sieve* factorisation method which works for all numbers.

See section D.2.

**Legendre's Symbol**

The value of *Legendre's symbol*  $(a/p)$  is defined to be +1, iff  $a$  is a quadratic residue of the odd prime  $p$ . If  $a$  is a quadratic non-residue of  $p$ , then  $(a/p) = -1$  and  $(a/p) = 0$  iff  $a$  is a multiple of  $p$ .

*Legendre's symbol* is only defined in the case  $p$  is an odd prime.

See section C.1.

**mod**

The relation  $a \equiv b \pmod n$  is read “ $a$  is congruent to  $b$  modulo  $n$ ” and means that  $a$  and  $b$  give the same rest when divided by  $n$ . Ex.:  $15 \equiv 1 \pmod 7$ , because  $15 = 2 \cdot 7 + 1$ . Often, one would like to minimize  $b$  to belong to the interval  $[0; n-1]$ .

**Multiple Polynomial Quadratic Sieve (MPQS)**

The *multiple polynomial quadratic sieve* is a version of the *quadratic sieve* factorisation method in which many polynomials are used to generate suitable sieving intervals.

See section 2.1.1.

**Number Field Sieve (NFS)**

The *number field sieve* is a factorisation algorithm similar to the *quadratic sieve*. The difference is that we have algebraic numbers of the number field in question instead of ordinary numbers. They are later transformed into ordinary numbers. Often, the term *number field sieve* refers to the *special number field sieve*.

See appendix D.

**Probabilistic Algorithm**

An algorithm is *probabilistic*, if it is not deterministic but rather guessing the result. The designation supposes that there is a good chance of success when applying the algorithm. This may not always be the case for every input. The algorithm may run forever or terminate with the conclusion that there is no result to be found with the current choice of parameters.

### **Quadratic Residue**

If  $a \equiv x^2 \pmod n$  and  $\gcd(a, n)=1$ , then  $a$  is called a *quadratic residue* of  $n$ . For more information, see section C.1, [1] or [2].

### **Quadratic Sieve (QS)**

The *quadratic sieve* is a factorisation algorithm that systematically builds a congruence  $x^2 \equiv y^2 \pmod N$  to find a factor of  $N$  by taking  $\gcd(x - y, N)$ .

See chapter 2.

### **RSA**

The *RSA* algorithm is a security algorithm invented in 1977 by Rivest, Shamir and Adleman. It bases on the difficulty of factoring large numbers.

See [11].

### **Sieve of Eratosthenes**

The *sieve of Eratosthenes* is a method to detect prime numbers. It works by crossing out all multiples of the prime numbers found so far, leaving the smallest non-crossed number as the next prime number.

### **Smooth Number**

An integer is called *smooth*, if it is a product of only small prime factors. In addition, if it has no prime factor  $>k$ , it is called *k-smooth*.

### **Special Number Field Sieve (SNFS)**

The *special number field sieve* is a version of the *number field sieve* factorisation algorithm that can only be applied to numbers that can be written in a special way.

See appendix D.

### **UML**

*UML* stands for *Unified Modelling Language* and is an open method used to specify, visualise, construct and document the artefacts of an object-oriented software-intensive system under development.

## 2 The Quadratic Sieve

---

This chapter contains a description of the quadratic sieve method. Necessary theoretical background is provided in appendix C.

Some other factorisation methods are presented in appendix A and appendix B. The number field sieve is described in appendix D and the elliptic curve method in appendix E. Since the thesis aims at developing a specific system for factorisation, not all available theory is presented in this document. For example, no algorithm for primality testing is given.

The interested reader is referred to [1] and [2] for a more profound introduction to the subject and to any of the papers given in chapter 8 for specifics on a certain method.

### 2.1 The Method

If we want to factor an integer  $N$ , we can look for integers  $x$  and  $y$  satisfying the equation  $x^2 - y^2 = N$  (as explained in section A.2), because  $x^2 - y^2 = (x + y) \cdot (x - y) = N$ , which is the product of two integers. That was already known to Fermat. The problem is to find suitable  $x$  and  $y$ . Maurice Kraitchik suggested that one could look for any  $x$  and  $y$  with  $x^2 \equiv y^2 \pmod{N}$  instead (see section A.3). That gives a 50% chance that  $\gcd(x - y, N)$  or  $\gcd(x + y, N)$  reveals a non-trivial factor of  $N$ . We no longer have a deterministic algorithm, but if we find several values for  $x$  and  $y$ , we have a very good chance of getting a factor. With that aim, the quadratic sieve method (QS) developed by Carl Pomerance proceeds as follows:

- Take  $m = \lfloor \sqrt{N} \rfloor$ .
- Set  $r_i = m + i$  starting with  $i = 1$  and define  $f(r_i) = r_i^2 - N$ .
- Now, search for possible prime factors  $p$  of  $f(r_i)$  by determining the value of the Legendre's symbol  $(N/p)$ , since  $N \equiv r_i^2 \pmod{f(r_i)}$  means that  $N \equiv r_i^2 \pmod{p}$  if  $p$  divides  $f(r_i)$ . Therefore,  $N$  must be a quadratic residue of  $p$ , i.e.  $(N/p) = +1$ . Set an upper bound for  $p$ .

**Definition 1:** The set of primes which are used for factoring  $f(r_i)$  is called the **factor base**.

- We know that if  $p^{\alpha_i}$  divides  $f(r_i)$  then  $p^{\alpha_i}$  divides  $f(r_i + k \cdot p^{\alpha_i})$ , too. This way, we can generate new  $f(r_i)$ s the same way as we generate new composite numbers in the sieve of Eratosthenes. This is the sieving part of the method.

- To determine, which  $f(r_i)$  are divisible by  $p^{\alpha_i}$  in the previous step, find  $\pm r_i \pmod{p^{\alpha_i}}$  such that  $r_i^2 \equiv N \pmod{p^{\alpha_i}}$ .
- Given all the factorisations of adequate  $f(r_i) = \prod_j p_j^{\alpha_{ij}}$ , where  $p_j$  belongs to the factor base, construct a binary  $i \times j$  matrix containing elements  $a_{ij}$  with  $a_{ij} = 1$ , iff  $\alpha_{ij}$  is odd.
- Use a matrix elimination method to search for a row with zeros. For that particular combination of  $f(r_i)$ s, all the exponents are even.
- Call the indices emerging from the previous step  $l$ .  
Then,  $\prod_l r_l^2 \equiv \prod_l f(r_l) \pmod{N}$ , where both sides are squares.
- We now have our sought congruence  $x^2 \equiv y^2 \pmod{N}$  and can easily calculate  $\gcd(x \pm y, N)$ . There is however a small risk that for example  $x = y$  or  $\gcd(x \pm y, N) = N$ . All we need to do then is to go back two steps and search for a new combination. (If we are unlucky we need to factor some more function values.)

### 2.1.1 Improvements

It can be difficult to find  $f(r_i)$ s which factor completely over the factor base. The solution is to allow separate large prime factors. The chances that another  $f(r_i)$  has the same large prime factor are good and when an even number of such  $f(r_i)$ s are combined, the large factor appears with an even exponent and does not need to be considered in the matrix.

Another improvement suggested by Peter L. Montgomery is to replace the function  $f(r_i)$  by polynomials  $F(x) = ax^2 + 2bx + c$  with  $N = b^2 - ac$ , so that  $a \cdot F(x) = (ax + b)^2 - N$ . As before, if  $p$  divides  $a \cdot F(x)$  then  $(N/p) = +1$ .

To keep  $F(x)$  as small as possible (and thus likelier to factor over the factor base), we want to choose  $a$ ,  $b$  and  $c$  so as to minimise both  $-F(-b/a) = N/a$  and  $F(M - b/a) = a \cdot M^2 - N/a$  (where  $M$  is half of the sieving interval).  $a$  should consequently be close to  $\sqrt{2N}/M$ . Choose  $a$  as the square of a prime,  $b$  so that  $b^2 \equiv N \pmod{a}$  and  $c$  as  $(b^2 - N)/a$ .

The final relation is  $\prod (ax + b)^2 \equiv \prod F(x) \pmod{N}$ .

This version of the quadratic sieve is called multiple polynomial quadratic sieve (MPQS) and will be the method implemented in the system described in this thesis.

## 2.2 Implementation

The description presented here follows the article [29] by Robert D. Silverman (except section 2.2.2, section 2.2.3 and section 2.2.4). There are several sieving methods one can implement, for example the lattice sieve proposed by John M. Pollard. In this particular system however, we only implement a simple sieving strategy as it was implemented in the early 90's.

Regarding the matrix elimination, we need to find a better method than simple Gaussian elimination since that would require too much memory. The block Lanczos method (see [27]) will be implemented instead.

### 2.2.1 Sieving

The following steps explain the implementation of the sieve:

- First, choose a polynomial to sieve.
- Set up an array of size  $2M + 1$  for the function values.
- Initialise it with *sieve locations*  $s_i = \log B_2 - \log(M\sqrt{N/2}) + B_3$  with  $B_2$  as the large prime bound,  $B_3$  empirically determined (small) and  $x = i - M$  for index  $i$ .
- For all primes  $p$  in the factor base (and for the prime powers, too), determine the function values that are divisible by  $p$ . For those function values, add  $\log p$  to the *sieve location*. As explained in section 2.1, we need to search for  $\pm t$  such that  $t^2 \equiv N \pmod p$ . Then,  $F(x)$  is divisible by  $p$  for  $x \equiv (\pm(t - b))/a \pmod p$  and for any other  $x' = x + kp$  with integer  $k$ .
- If, after the sieving  $s_i \geq 0$ , it is likely that the corresponding  $F(x)$  is  $B_1$ -smooth (where  $B_1$  is the prime bound for the factor base) with the exception of a large prime factor below  $B_2$ .
- Factor those function values using trial division.
- Sieve with other polynomials until the number of smooth function values exceeds the number of primes in the factor base.

### 2.2.2 Lanczos Method

Given a symmetric, positive definite  $n \times n$  matrix  $A$  with real elements, Lanczos method solves the equation  $Ax = b$  for a vector  $b$ .

- Define a sequence of vectors  $w_i$  recursively as:

$$w_0 = b, \quad w_i = Aw_{i-1} - \sum_{j=0}^{i-1} c_{ij}w_j \quad \text{for } i > 0 \quad \text{and} \quad c_{ij} = \frac{w_j^T A^2 w_{i-1}}{w_j^T A w_j}.$$

The  $c_{ij}$  are chosen so that the vectors are orthogonal with respect to  $A$ .

- After at most  $n$  iterations, a zero vector will appear for  $w_m$ . This is partly because  $n + 1$  vectors are linearly dependent (see [27]).

- Then,  $x = \sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} w_j$  is the solution to  $Ax = b$ .

According to [27],  $c_{ij} = 0$  for  $j < i - 2$ , so that the expression simplifies to  $w_i = Aw_{i-1} - c_{ii-1}w_{i-1} - c_{ii-2}w_{i-2}$  for  $i \geq 2$  (and, of course,  $w_1 = Aw_0 - c_{10}w_0$ ).

$$c_{ii-1} = \frac{(Aw_{i-1})^T (Aw_{i-1})}{w_{i-1}^T (Aw_{i-1})} \text{ and } c_{ii-2} = \frac{(Aw_{i-2})^T (Aw_{i-1})}{w_{i-2}^T (Aw_{i-2})}.$$

The running time of Lanczos method is order  $dn^2$ , if there are on average  $d$  non-zero elements per column in  $A$ .

### 2.2.3 Block Lanczos

This method is explained in [28].

Replace the Lanczos iterations by:  $W_i = V_i S_i$ ,

$$V_{i+1} = AW_i S_i^T + V_i - \sum_{j=0}^i W_j C_{i+1j} \text{ and}$$

$$C_{i+1j} = (W_j^T A W_j)^{-1} W_j^T A (AW_i S_i^T + V_i), \text{ all for } i \geq 0.$$

The construction of the matrices  $S_i$  will be explained later.

The final solution is obtained, when  $V_m^T A V_m = 0$  and  $V_m \neq 0$ :

$$X = \sum_{i=0}^{m-1} W_i (W_i^T A W_i)^{-1} W_i^T V_0.$$

Almost the same simplification as in standard Lanczos can be achieved:

$$V_{i+1} = AW_i S_i^T + V_i - W_i C_{i+1i} - W_{i-1} C_{i+1i-1} - W_{i-2} C_{i+1i-2}.$$

With the introduction of  $W_i^{inv} = S_i (W_i^T A W_i)^{-1} S_i^T$ , we get

$$V_{i+1} = AV_i S_i S_i^T + V_i D_{i+1} + V_{i-1} E_{i+1} + V_{i-2} F_{i+1}, \quad i \geq 0, \text{ where}$$

$$D_{i+1} = I_N - W_i^{inv} (V_i^T A^2 V_i S_i S_i^T + V_i^T A V_i),$$



$$E_{i+1} = -W_{i-1}^{inv} V_i^T A V_i S_i S_i^T,$$

$$F_{i+1} = -W_{i-2}^{inv} (I_N - V_{i-1}^T A V_{i-1} W_{i-1}^{inv}) \\ (V_{i-1}^T A^2 V_{i-1} S_{i-1} S_{i-1}^T + V_{i-1}^T A V_{i-1}) S_i S_i^T \quad \text{and}$$

$$W_i^{inv} = S_i (S_i^T V_i^T A V_i S_i)^{-1} S_i^T$$

with  $W_j^{inv} = 0$ ,  $V_j = 0$  and  $S_j = I_N$  for  $j < 0$ .

$$\text{Then, } X = \sum_{i=0}^{m-1} V_i W_i^{inv} V_i^T V_0.$$

About the matrices  $S_i$ :

$S_i$  is a matrix that chooses columns in the matrix multiplied with it. As a consequence,  $S_i S_i^T$  is a submatrix of  $I_N$ , i.e. it is an  $N \times N$  identity matrix with some additional zeros. Here is an algorithm for choosing the ones in  $S_i S_i^T$  and calculating  $W_i^{inv}$  according to [28]:

- Let  $T = V_i^T A V_i$  and  $S_{i-1}$  be given.
- Construct a matrix  $M$  with  $T$  on the left and  $I_N$  on the right.
- Number the columns of  $T$  as  $c_1, c_2, \dots, c_N$  with columns in  $S_{i-1}$  coming last.
- For all columns  $c_j$  do the following:
  - Search for the first row “below and including  $c_j$ ” (that is a row with a higher or equal index  $c_k$ ) where the element in column  $c_j$  is not zero. If such a row is found and it is not row  $c_j$  itself, exchange the two rows.
  - If now the element at position  $[c_j, c_j]$  is not zero (meaning there is a pivot element in column  $c_j$ ), set a one at position  $[c_j, c_j]$  in  $S_i S_i^T$  and zero the rest of column  $c_j$  by row addition (which is done by a single exclusive-or operation since we are working *modulo 2*). If we were not working *modulo 2*, we would need to divide row  $c_j$  by  $M[c_j, c_j]$ .
  - If, on the other hand, no pivot element is found in column  $c_j$ , repeat the search for a non-zero element on the right hand side of the matrix (that is to say in column  $c_j + N$ ). Again, exchange the two rows. By construction, there must be such an element, so that we can assert that  $M[c_j, c_j + N]$  is not zero. Zero out the rest of column  $c_j + N$  by row addition and then zero row  $c_j$  of  $M$ .
- When the algorithm is done,  $W_i^{inv}$  can be found in the right half of  $M$ .

### 2.2.4 Application of Lanczos on MPQS

The problem with our matrices is that they are neither square nor positive definite, so we cannot use standard Lanczos. The block Lanczos algorithm described by Peter L. Montgomery in [28] as described above overcomes these difficulties. The non-squareness is tackled by solving  $(B^T B)X = 0$  instead of  $BX = 0$ . Now we have an  $A$  for the algorithm, but if we tried solving  $AX = 0$ , we would only get the trivial solution. Therefore, choose a random vector  $Y$  of size  $n \times N$ , where  $N$  is the size of one word. Choose  $V_0$  as  $AY$  and solve  $AX = AY$  with block Lanczos. Use  $X - Y$  to find the vectors we need.

This is done in the following way:

- Form  $Z$  as the columns of  $X - Y$  concatenated with those of  $V_m$ .
- Compute  $BZ$  and find a matrix  $U$  whose columns span the null space of  $BZ$ .
- Output a basis for  $ZU$ .

We should avoid computing with matrices of size  $n \times N$ , since  $n$  can get very large. Also, we should not store unnecessarily many temporary matrices. However, we can afford storing some extra matrices of size  $N \times N$ , which fit into  $N$  words each, if there are other benefits. To avoid calculating  $X - Y$  at the end of the algorithm, we can keep track of the partial sums. As an improvement,

we can use  $V_{i+1}^T V_0 = D_{i+1}^T V_i^T V_0 + E_{i+1}^T V_{i-1}^T V_0 + F_{i+1}^T V_{i-2}^T V_0$

for  $i \geq 2$ , where  $V_{i-k}^T V_0$  for  $k = 0, 1, 2$  are known by induction.

### 2.2.5 Choice of Parameters

Robert D. Silverman suggests multiplying  $N$  by a small constant  $k$ , such that  $kN \equiv 1 \pmod{8}$ , because then 2 is in the factor base.

The prime bound  $B_1$  for the factor base should be chosen asymptotically about  $\exp((1/2 + o(1))\sqrt{\log N \log \log N})$  according to [27]. The running time of the quadratic sieve is then (again, according to [27])  $\exp((1 + o(1))\sqrt{\log N \log \log N})$ .

$M$  can be chosen fairly small, since we want to have values likely to be smooth and we can always choose more polynomials. However, one does not want to waste too much time choosing new polynomials.

### **2.2.6 Efficiency Considerations**

Sieving should not be done on the entire array at once (see [27]). Instead, the array should be split into blocks which are sieved completely before going to the next block. For the distributed part of the system, this means that different blocks can be allocated to different clients. Moreover, different polynomials can of course be distributed to different clients.



## 3 Implementation Details

---

In this section, the details of the implementation are introduced.

The programming language will be C/C++ throughout the thesis. There are good chances for a fast system with that choice of language. C/C++ provides all benefits of a high-level language and can easily be complemented by macros written in an assembly language when necessary.

### 3.1 LIP

The Long Integer Package (LIP) by Arjen K. Lenstra will be used for handling long integers. It is implemented in C and is intended for non-commercial use. See [12].

Examples of what it can do:

- Perform arithmetic operations on large numbers.
- Generate small prime numbers.
- Primality testing.
- Calculate  $\gcd(a, b)$ .

asf.

Something else that can be useful in this package is the Montgomery modular arithmetic invented by Peter L. Montgomery. It was designed to do division free modular multiplication and this is about 20% faster than ordinary modular multiplication.

The package also contains some factorisation algorithms which will not be used for this system.

The timing utilities will be used to evaluate test runs.

### 3.2 LiDIA

LiDIA is a C++ library for computational number theory. It was developed and is maintained by the LiDIA group at the Darmstadt University of Technology (Germany). Like LIP, it is intended for non-commercial use and can be downloaded at their homepage (see [13]).

Examples of what it can do:

- Handle elliptic curves.
- Represent higher degree number fields.

- Determine roots of a polynomial (modulo  $p$ ).
- Factor ideals of algebraic number fields.

asf.

This package could be used as an aid for implementation of the number field sieve and the elliptic curve method. Unfortunately, it never came into play in this system due to lack of time. I recommend it for future work on the system.

### 3.3 Other Software

The server program needs to store information about users and data on currently running factorisations. This will be done in a database. Therefore, the system needs a DBMS (database management system) and ODBC (open database connectivity) drivers.

Seeing that there already is a DBMS installed on the Mandrake system mentioned in section 3.4, it can be used without further ado. The DBMS in question is called PostgreSQL, it is OpenSource and can be downloaded at the PostgreSQL web page (see [15]). The installed version is 7.3.4.

Note that the DBMS can be replaced with hardly any modifications to the source code since ODBC is completely transparent. The only thing that maybe would need to be replaced is the connection call. The installed ODBC driver/driver manager at the server side is unixODBC version 2.2.6-4mdk (see [16] for more information).

It is possible to implement the database interface without the use of further libraries, but I chose to use a library called libodbc++ instead (see [17]). It is an OpenSource project just like the above projects and the installed version is 0.2.3. It provides a subset of the well-known JDBC (Java database connectivity) and the database interface becomes easier to implement, to read and less likely to contain errors than without it.

Furthermore, it is necessary for the server to be able to send e-mail messages since the users of the system might forget their password and want to have it sent to them. For that purpose, the sendmail program (version 8.12.9) will be used (see [18]) which will submit e-mail messages to the SMTP server installed on the Linux/Mandrake system. Note: That server is not accessible for the public.

### 3.4 Environment

The algorithms and the server program should be able to run on various Unix platforms. Binaries will be compiled for Linux on a PC and for Solaris on a Sun server (see table 1 for details).

The client program should additionally be able to run on Microsoft Windows, because that is the prevailing operating system among home users. A binary for the client program will be compiled for Windows and for the previously mentioned operating systems.

Also, there should be test runs for the compiled binaries (algorithms only, server program and client program).

The compiler that will be used is gcc 3.3.1 in Mandrake, gcc 3.3.2 in Fedora, gcc 2.95-4 in Debian and gcc 2.95.3 in Solaris. In Windows, Cygwin version 1.5.5-1 will be used in combination with gcc 3.3.1-3 (see also [14]). For test purposes, the program will also be compiled on IRIX64 with the MIPSpro C/C++ compilers. Also, there will be a comparison with Portland Group's (PGI) compilers pgcc and pgCC (see [19]) and Intel's compiler icc (see [20]).

In case a debugger is needed, it will be gdb 5.3-25 in Mandrake. For memory debugging, a program called Valgrind (see [21]) will be used. Version: 2.0.0 in Mandrake.

To make development of the source code easier and safer, the version control system CVS (Concurrent Versions System) will have its eyes on it (see [22]). The source code will be written using a standard editor. A special development kit should therefore not be necessary at all.

Operating System	Details	Platform	Processor	Memory
Linux/ Mandrake	Mandrake 9.2 (FiveStar) Kernel: 2.4.22- 10mdk	PC, i686	Intel Pentium 4 2.4 GHz 512 KB cache ca. 4797 Mips	1 GB RAM  620 MB swap
Microsoft Windows	Windows 2000 Professional and/or Windows XP Professional	see above	see above	see above *

Table 1: Available Compilation/Runtime Environment.

Operating System	Details	Platform	Processor	Memory
Linux/ Fedora	Fedora Core release 1.90 (FC2 Test 1) Kernel: 2.6.3-1.97	PC, i686	Intel Pentium 4 3.2 GHz 512 KB cache ca. 6324 Mips	1 GB RAM  1 GB swap
Linux/ Debian	Debian GNU/Linux 3.0 (woody) Kernel: 2.4.18-1-k7	PC, i686	AMD Athlon 1.2 GHz 256 KB cache ca. 2385 Mips	256 MB RAM  977 MB swap
Unix/Sun Solaris	Sun Solaris 8 running SunOS 5.8	Sun server	UltraSPARC-IIi 440 MHz 2 MB cache ca. 865 Mips	512 MB RAM  5.5 GB swap
Linux/ Red Hat	Red Hat Linux 7.2 (Enigma) Kernel: 2.4.20-24.7	PC, i686	AMD Athlon XP 1800+ 1.53 GHz 256 KB cache ca. 3061 Mips	512 MB RAM  1 GB swap
Linux cluster/ Red Hat	Red Hat 7.3 (Valhalla) Kernel: 2.4.18-27.7 .xsmmp-cap1	198 x PC, i686 + 6 login nodes	396 x Intel Xeon 2.2 GHz 512 KB cache ca. 4381 Mips	2 GB RAM/PC 80 GB swap/PC
IRIX	IRIX (64 bit) version 6.5.22f	SGI Origin 3800	128 x MIPS R14000 500 MHz 8 MB cache max 1 GFlop	1 GB RAM/pro-cessor + 128 GB shared

Table 1: Available Compilation/Runtime Environment.

\* Note: Since Windows uses its own partition for swap, it is dependent on free available disk space. In this case, there should be at least several gigabyte available on the smaller partition and the program should never run out of memory.



### 3.4.1 nsieve Benchmark

The nsieve program runs the sieve of Eratosthenes for different array lengths and it can be downloaded at [23]. It calculates two rates: High MIPS and Low MIPS. The high rate was calculated for an array of 2,560,000 bytes and the low rate was calculated for an array of 8191 bytes. Their value should not be confused with the proper Mips rate according to the traditional definition as million instructions per second. But seeing that the system for factorisation will implement sieving methods, they seem like adequate comparison rates. Like intuition suggests, a better rate means faster computation.

The results of the benchmarking test are displayed in table 2.

Operating System (as above)	High MIPS	Low MIPS
Linux/Mandrake	1414.8	150.6
Linux/Fedora	1865.9	252.5
Linux/Debian	1317.2	80.5
Unix/Sun Solaris *	ca. 483.9	ca. 98.4
Linux/Red Hat *	ca. 1476.3	ca. 137.0
Linux cluster/Red Hat	1300.4	180.9

Table 2: nsieve Benchmarking Results.

\* Note: The Solaris and the Red Hat system are multi user environments and the existing computer load can fluctuate. This is true for the benchmarking test as well as for the actual test runs of the system. It means that we probably will not have access to the maximum available CPU time/cache and RAM according to table 1. The benchmarking test gave varying results of which the table shows the respective top result.



## 4 Design

---

This chapter describes the design of the system in general and the client and server application in particular. It also defines the network protocol that is used for client/server communication.

The thesis only applies to the quadratic sieve algorithm because there was no time to implement other methods (see chapter 6). However, it should be easy to add new methods.

### 4.1 Code Structure

Since the implementation is done mostly in C++, it is appropriate to sketch the overall object structure of the code here to gain an overview of the system's design.

Before I began developing the distributed part, I made a working program that can read parameters from a file, sieve and output the obtained relations in another file. That was/is the “standalone application”.

#### 4.1.1 Standalone Application Structure

As mentioned before, the standalone application reads the factorisation parameters, does all the sieving and writes the result to a file. Unfortunately, the implementation remains unfinished and a final result is not always found.

As figure 1 shows, the object *QuadraticSieve* is responsible for all of the sieving and for coordination of the other objects. Based on the input, it generates a factor base which is stored in a *QSFactorBase* object consisting of *QSFactorBasePrime* objects. Then, it does the sieving. Partial relations that are found are temporarily stored in *QSRelation* objects. By and by, they are combined to full relations as new partial relations with the same large factor occur. Full relations and merged relations are immediately written to the output file.

After the sieving, *QuadraticSieve* creates a *QSBlockLanczos* object that does matrix elimination. The *QSBlockLanczos* reads the file where the relations were stored and makes a *QSSparseMatrix* object of it. It initialises all of the other matrices required by the algorithm and does its calculations with the help of *QSMatrix* and *QSIdentitySubMatrix* objects. The iteration stops when a result matrix is found. That matrix is returned to *QuadraticSieve* which tries to find a suitable congruence according to section 2.1 and

divides out the found factor.

The reason for having a separate class *QSFinalRelation* where the exponents of the right hand side of the final congruence are accumulated, is that there are potentially many more distinct prime factors involved than in a single function value factored in the sieving step. So we can allocate less space for every *QSRelation* than for every *QSFinalRelation* without risking not having enough memory to compute the final congruence.

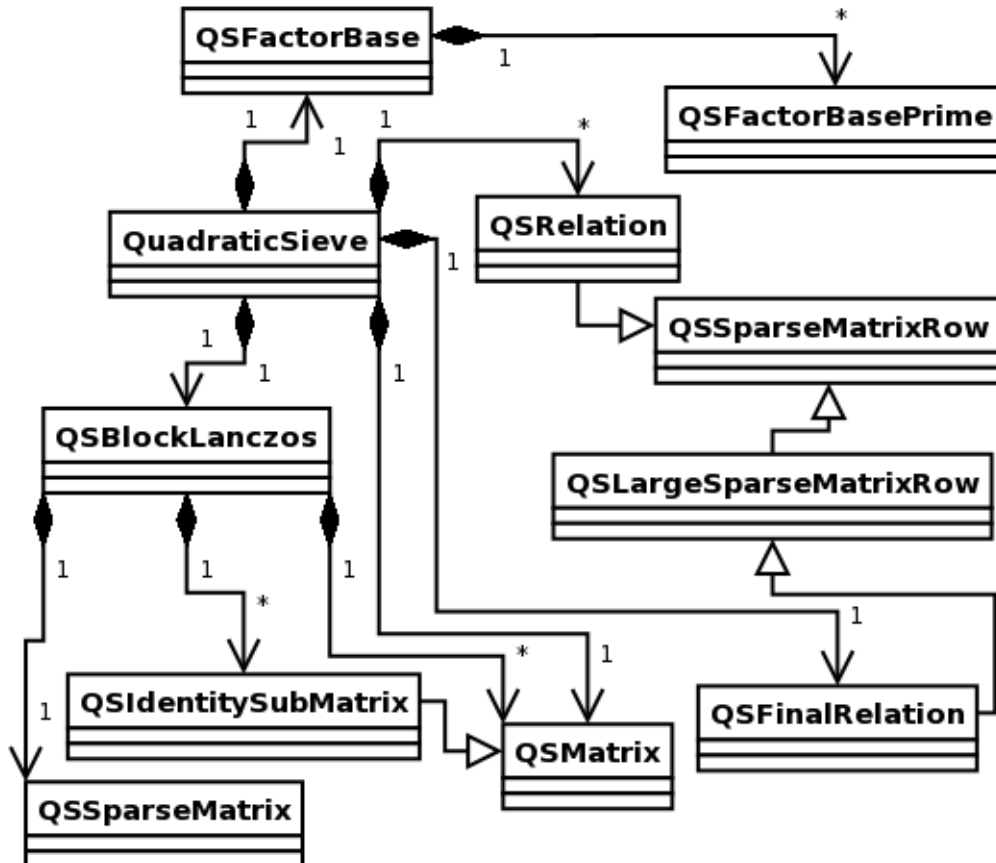


Figure 1: UML Diagram of the Standalone Application.

In the source code, the class declarations/definitions are grouped as follows: the *QuadraticSieve* files (*.h/.cpp*) contain *QuadraticSieve*, the *QSFactorBase* files contain *QSFactorBasePrime* and *QSFactorBase*, the *QSMatrix* files contain *QSSparseMatrixRow*, *QSRelation*, *QSLargeSparseMatrixRow*, *QSFinalRelation*, *QSMatrix*, *QSIIdentitySubMatrix* and *QSSparseMatrix* and the *QSBLOCKLanczos* files contain *QSBLOCKLanczos*.

Additional files needed for compilation: *main.cpp* which contains the main program, *definitions.h* contains some global definitions, *params.txt* which contains the factorisation parameters and the files *lippar.h*, *lip.h* and *lip.c* that contain the LIP (see section 3.1).

### 4.1.2 Distributed Application Structure

The main difference between the standalone application structure and the distributed application structure is the latter's decentralisation of the sieving step. Sieving is no longer concentrated to one single object. As explained in section 3.3, this brings forth the need of a means to make parameters and intermediate results available to the users.

This is accomplished by storing all vital parameters and data in a database at the server side and then negotiate with the clients via network.

Unfortunately, also the server and client remain unfinished. The sieving part of the program is rewritten and its structure is depicted in figure 2. There is also a server utility which helps creating a new project and putting the necessary data into the database (called createProject). The server part is inchoate. Its structure so far can be seen in figure 3. And finally, the client part is not started upon, although most of the job there would be to combine the sieving part with some network classes closely related to those in the server.

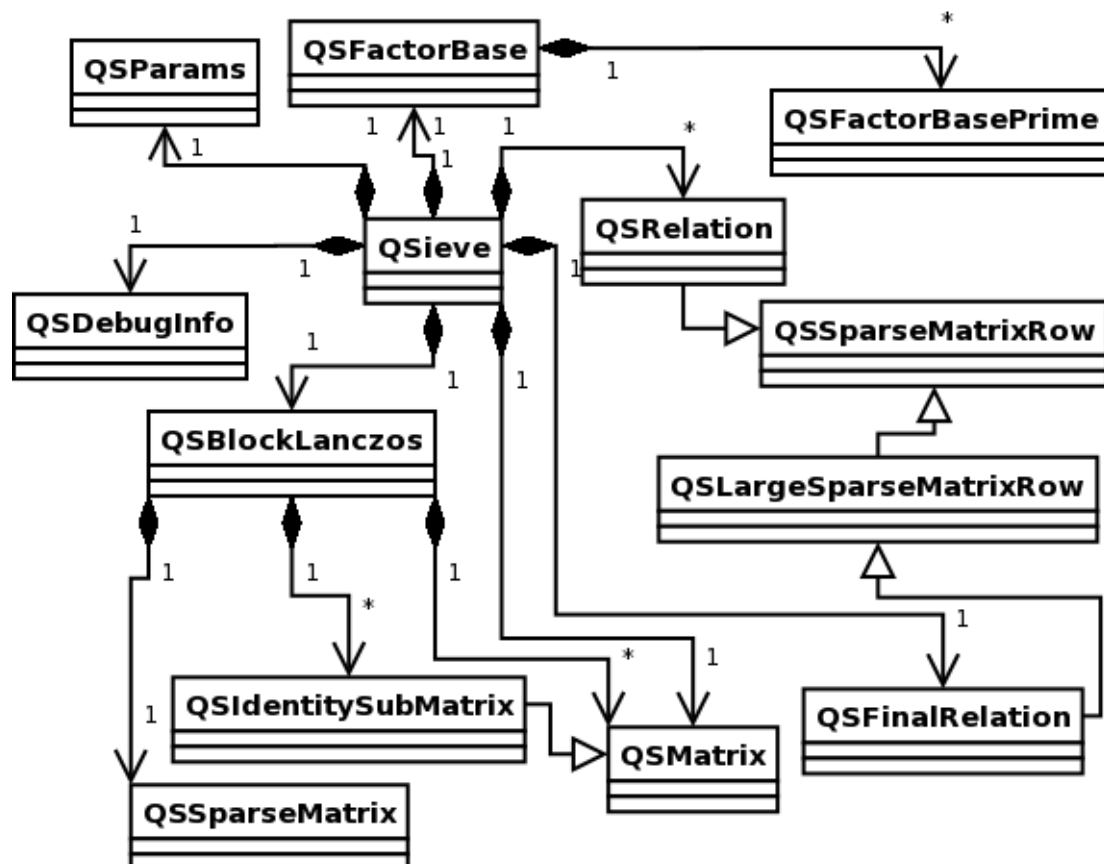


Figure 2: UML Diagram of the Restructured Sieving Part.

The new structure of the sieving part is not that different from the old. The only two objects that are new are *QSPams* which contains the parameters of the quadratic sieve and *QSDebugInfo* which holds debugging information like various time variables, the number of polynomials generated and the number of smooths found. These components were already present before, only located in the *QuadraticSieve* object itself instead.

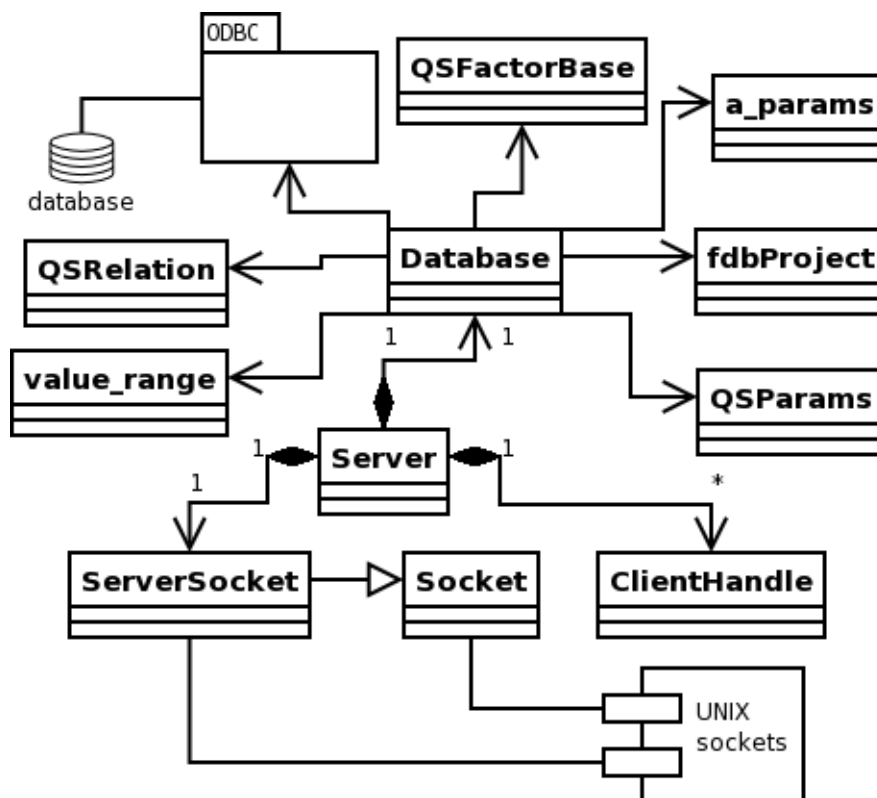


Figure 3: UML Diagram of the Server Part.

The server part looks as follows: The server object *Server* is responsible for starting up the server and connecting to the database. For that purpose, it has a *ServerSocket* object and a *Database* object. The *ServerSocket* class is a subclass of *Socket* and both use the socket API provided by the operating system. The *Database* object runs its queries to the database through the ODBC API (see section 3.3). It also makes use of various classes for storing and retrieving data. Once the server is up and running, it listens to client connections and when a connection is established successfully, it creates a new *ClientHandle* object for the client and stores it in a vector.

New files: the *Socket* files (*.h/.cpp*) contain *Socket*, *ServerSocket* and *ClientHandle*, the *Database* files contain *Database*, the *Server* files contain *Server*, the *Datatypes* file (*.h*) contain *fdbProject*, *a\_params*, *value\_range*, *QSPams* and *QSDebugInfo* and the *QSieve* files contain *QSieve*.

## 4.2 Data Structure

It is not necessary nor advantageous to list all the internal data structures of the system here. However, it can be useful to depict the structure of the data that is visible/accessible to the user in the distributed application.

Seeing that all data is stored in a database at the server side, the best way to show the data structure is via an ER-diagram.

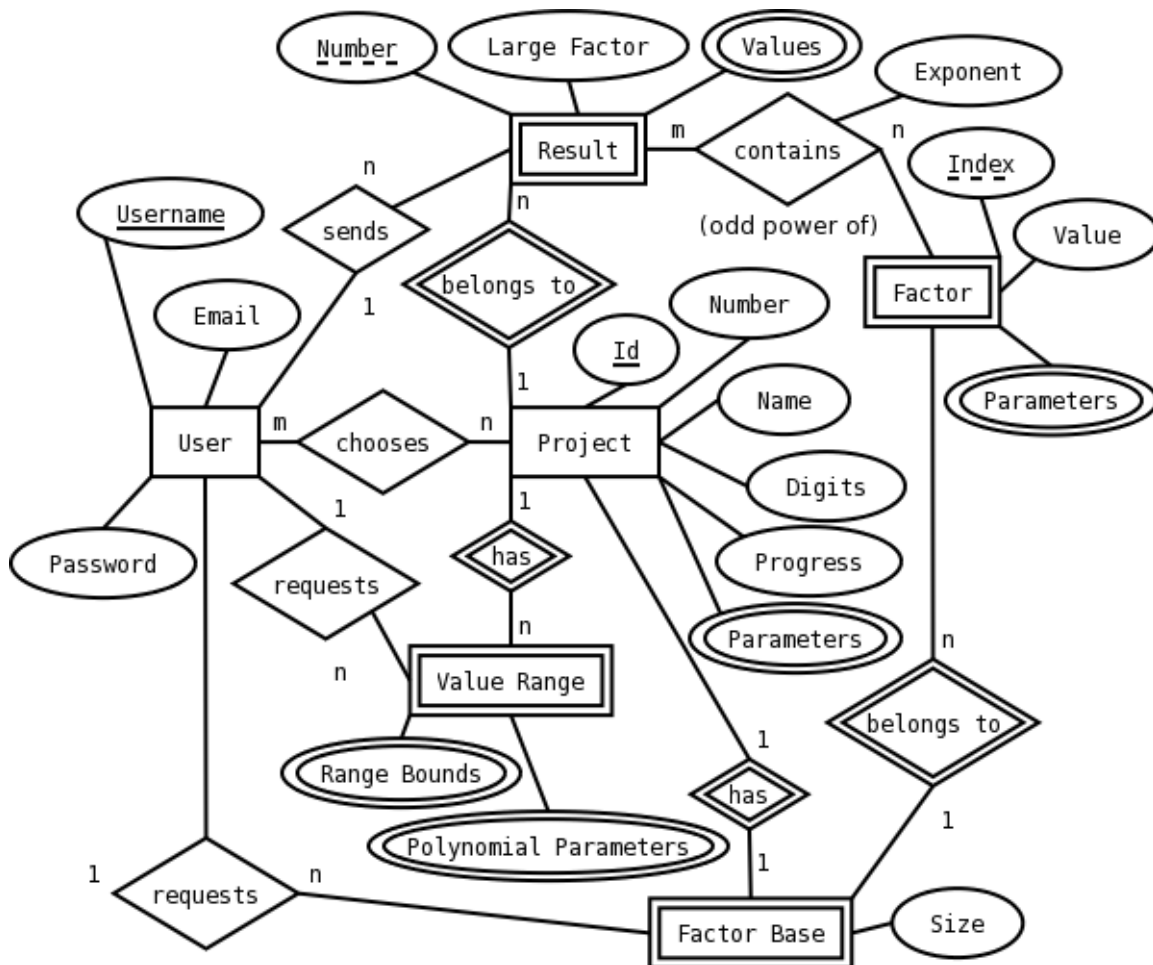


Figure 4: ER-diagram for the database.

The design in figure 4 seems complicated and cumbersome. This is due to the way the user interacts with the data in the system and how that data is transferred between client and server. Essentially, the structure is actually quite simple: On the one side, there is a user (having a user name, a password and an email address) and on the other there is a factorisation project. The project contains an id number, the number to be factored, possibly a name associated with the number to be factored, the number of digits of the number to be factored and the progress of the factorisation. The

progress is measured as how many percent of the required number of relations have been found.

To be able to factor the number, we also need a factor base associated with the project. Data about the factor base (with all included factors of course) must be transferred to the client. As for now, this will be done every time the client requests a value range and the two relationships labelled “requests” in the diagram are in fact only one request. A value range in turn is a sieving interval distributed to the client. It consists of the polynomial parameters and the block number. When a project is initiated, there are no value ranges at first. They are created by and by as clients send their requests and until the number can be factored (i.e. enough relations have been found).

The last thing that is part of the diagram is the term result. A result message typically consists of the relations (full and partial) found in the sieving step at the client. Hence, one single result which is part of the overall result is equivalent to a relation. For identification purposes (and for the purpose of counting relations), every result is associated with a number.

The multivalued attributes labelled “Parameters” in the diagram represent some extra internal parameters which are stored for the sieving algorithm.

### 4.3 Network protocol

The network protocol used for communication between server and client in this system is placed on top of TCP/IP. It is comparable with other simple protocols, such as SMTP and gives minimal security and reliability in present condition. Each message consists of a message code and a message text. The message text itself is the name of the instruction (which must match the message code) and possibly some additional data.

The previous section introduced the data structures that are involved in any interaction with the user and thus need to be subject to negotiations with the server. The task of the protocol is to provide instructions to control the creation, modification and transfer of such data. At present, there are no instructions for deletion. In addition to instructions related to data (codes 200-899), there are some communication control messages (codes 100-199) and error messages (codes 900-999). All presently used messages are listed in table 3.



The reason for splitting the user data into its elements is that there are many different situations where user data is sent/received and accordingly, many combinations of its elements occur.

<b>Message Code</b>	<b>Instruction</b>	<b>Additional Data</b>	<b>Purpose</b>
100	HELLO		Client requests connection.
110	READY		Instruction received successfully, ready for more input or ready to send data.
120	CONFIRMED		Instruction received successfully, something has been performed/saved.
130	FINISHED		End of data.
210	USER CREATE		Client request to create a new user.
220	USER LOGIN		Client request to log in a user.
250	USER	user name	Transfer of user name.
310	PASS REQUEST		Client requests that the server sends a password via email.
320	PASS CHANGE		Client request to change a password.
350	PASS	password	Transfer of password.
410	EMAIL CHANGE		Client request to change an email address.
450	EMAIL	email	Transfer of email address.
510	LIST		Client requests a listing of ongoing factorisations.
650	PROJECT	project	Transfer of factorisation project.
710	RESULT MESSAGE		Client requests submission of a result.
750	RESULT	result	Transfer of result data.

**Table 3: Messages Included in the Network Protocol.**

<b>Message Code</b>	<b>Instruction</b>	<b>Additional Data</b>	<b>Purpose</b>
810	VALUES REQUEST	project id	Client requests a value range for factorisation.
850	VALUES	values	Transfer of a value range and sieving information.
900	SERVER ERROR		Some error at the server (may have various causes).
905	SOCKET CLOSED		The socket was closed. Used internally at the server side.
911	DATA INVALID		Error in data transfer, invalid data received.
921	USER INVALID		Error when a user name is empty.
922	USER UNKNOWN		Error when logging in or requesting a password with a non-existing user name.
923	USER EXISTS		Error when creating a user with an existing user name.
931	PASS INVALID		Error when setting a password that does not meet password requirements.
934	PASS MISMATCH		Error when a received password does not match the stored one for the user.
941	EMAIL INVALID		Error when an email address is empty.
962	PROJECT UNKNOWN		Error when server receives a non-existing project id.
991	CODE INVALID		Error when server/client gets a message with unexpected code.
992	CODE UNKNOWN		Error when server gets a message with non-existing code.

Table 3: Messages Included in the Network Protocol.

The usage of these messages is partly described in figure 5 and figure 6, which are flow chart diagrams of the client communication with the server. There is no reason for the server to contact the client, so there is no further communication. The server response is not explicitly written in the diagram, but it can be deduced from the available messages in table 3.

The flow chart diagram in figure 5 shows what a client can do when the user is not logged in. First, the client must connect to the server by establishing a connection and sending a HELLO message. The server replies with READY if all went well. Then, the client can send a request for a new user to be created with a USER CREATE message. The server would respond with a READY message and the client can send its USER message. The server should then send CONFIRMED, but it can also send USER INVALID, USER EXISTS or SERVER ERROR. If the user name was ok, the client can send the desired password. There are three possible responses from the server: CONFIRMED, PASS INVALID or SERVER ERROR. Finally, the client sends the email address and gets either CONFIRMED, EMAIL INVALID or SERVER ERROR back. Hopefully, the client succeeded in creating a new user which can now log in. This is done by sending a USER LOGIN message to the server. The server replies by sending a READY message and the client sends the user name in a USER message. It can get back CONFIRMED, USER INVALID, USER UNKNOWN or SERVER ERROR. Then, the client transfers the password (as it is now in plain text) and the server responds CONFIRMED, PASS MISMATCH or SERVER ERROR.

As a third possibility, the client can send a PASS REQUEST message. If the server sends back READY, the client sends the user name and upon success, gets back a CONFIRMED message. If that is the case, it means that the server sent the password to the stored email address.

Once the user has logged in, he/she has five options. The user can change his/her password or email address, he/she can request a listing of ongoing projects, request a value range for a specific project or submit a result.

Changing the password or email address is done in the following way: The client sends a PASS CHANGE/EMAIL CHANGE message and the server should respond with READY. Then, the new password or email address is sent to the server, which sends CONFIRMED, PASS INVALID/EMAIL INVALID or SERVER ERROR.

If an error occurs, the client is logged out automatically, but stays connected to the server. If all goes well on the other hand, the user is still logged in and can perform other operations.

When a user requests a list of ongoing projects, the client sends a LIST message to the server. It gets a READY message back and confirms that it is ready to receive the list with a READY message of its own. The server then sends a PROJECT message and if it arrives successfully at the client side, the client sends a CONFIRMED message. The server either transmits the next PROJECT message or terminates the list with a FINISHED message.

A request for a value range is negotiated as follows: The client sends a VALUES REQUEST message with the corresponding project id number. The server looks up the project and can respond with READY, PROJECT UNKNOWN or SERVER ERROR. In the first case, the client also sends a READY message and the server transmits a VALUES message. The client should respond with a CONFIRMED message and the server sends a final CONFIRMED. If the server does not receive the client's CONFIRMED message correctly, the user is logged out.

Submission of a result is initiated by a RESULT MESSAGE message from the client. If the server sends back a READY message, the client can transmit the first result in a RESULT message. Every RESULT message by the client is confirmed by a CONFIRMED message from the server. When the client has sent all the results, it sends a FINISHED message instead and the server completes the transmission with a CONFIRMED message.

At any point, server or client can send a CODE INVALID message when the received message does not have the expected code. Also, a DATA INVALID message can be sent when the code and data mismatch. At the beginning of an operation, the server can send a CODE UNKNOWN message if there is no such message code as the received one. Finally, a SERVER ERROR can have several different causes, for example that the connection to the database server broke down and the server needs to be restarted. Another example is that the user was logged out but the client tries to send result messages.

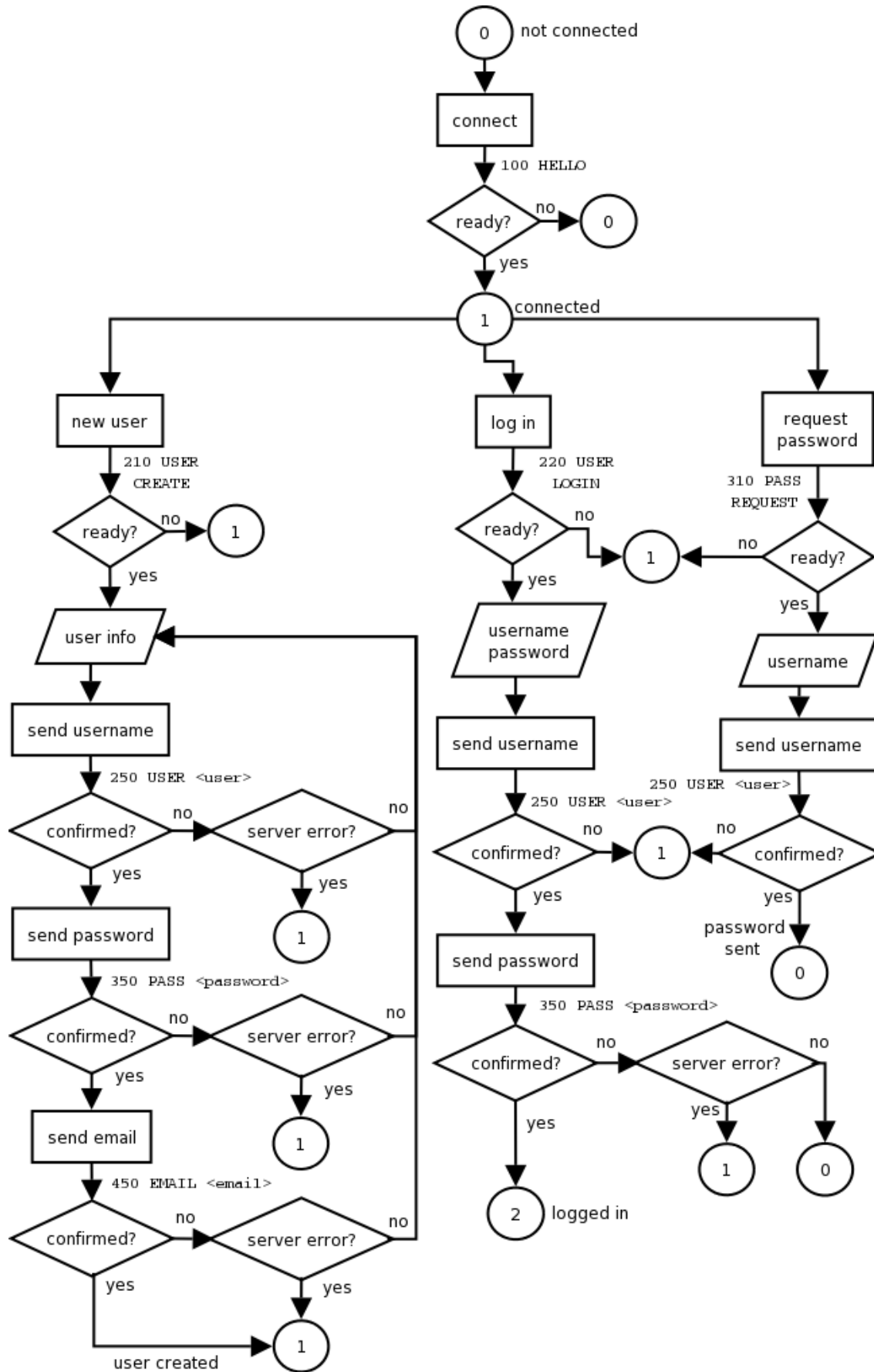


Figure 5: The Client's Flow Chart Diagram - Part 1.

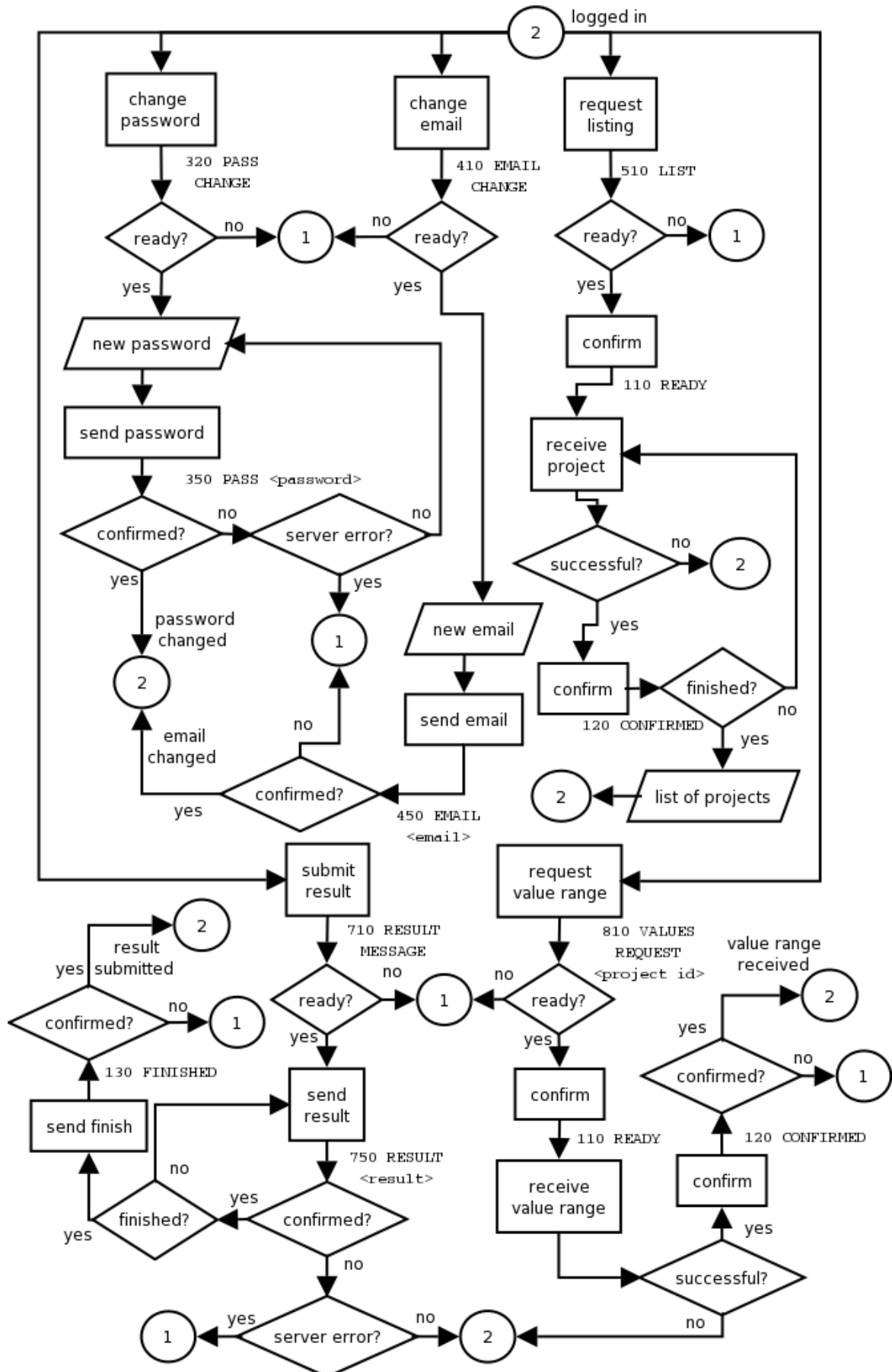


Figure 6: The Client's Flow Chart Diagram - Part 2.

## 5 Results

This chapter gives some results and observations from test runs. Unless stated otherwise, the results are from the unfinished standalone application run on the Linux/Mandrake system (see section 3.4). Observe that the running times of a working application would not differ much.

### 5.1 General Observations

This section examines the running times of the standalone application with respect to the size of the number to be factored. The test runs were performed on the Linux cluster/Red Hat system. The running times on that system are very exact, meaning that two runs with the same parameters will not differ noticeably in running time, because the calculation nodes in the cluster are reserved exclusively for the test run in question.

First, let us see what happens when we make use of Silverman's parameters from [29]. What will the results be?

Almost certainly, factoring larger numbers will take more time. The running time can be expected to grow exponentially. Also, running times are probably much lower nowadays than in Silverman's article.

The results of the test runs are as follows:

Size of the Number	M	Factor Base Size	Large Prime Bound	Running Time (Seconds)
24 digits	5,000	103	42,000	0.07
30 digits	25,000	206	141,000	0.25
36 digits	25,000	402	4,394,000	0.78
42 digits	50,000	907	256,000,000	9.83
48 digits	100,000	1234	$4.84 \times 10^8$	29.25
54 digits	250,000	2009	$1.19 \times 10^{10}$	162.17
60 digits	350,000	3076	$2.935 \times 10^{11}$	594.07
66 digits	500,000	4517	$8.28 \times 10^{12}$	3381.75

Table 4: Results of the First *Number Size Comparison* Test Runs.

Put into a diagram, it looks this way: The x-axis shows the size of the number, the primary y-axis shows the curve for the running time (solid line) and the secondary y-axis shows the curve for the sieving bound  $M$  (dashed line). Both y-axes have a logarithmic scale.

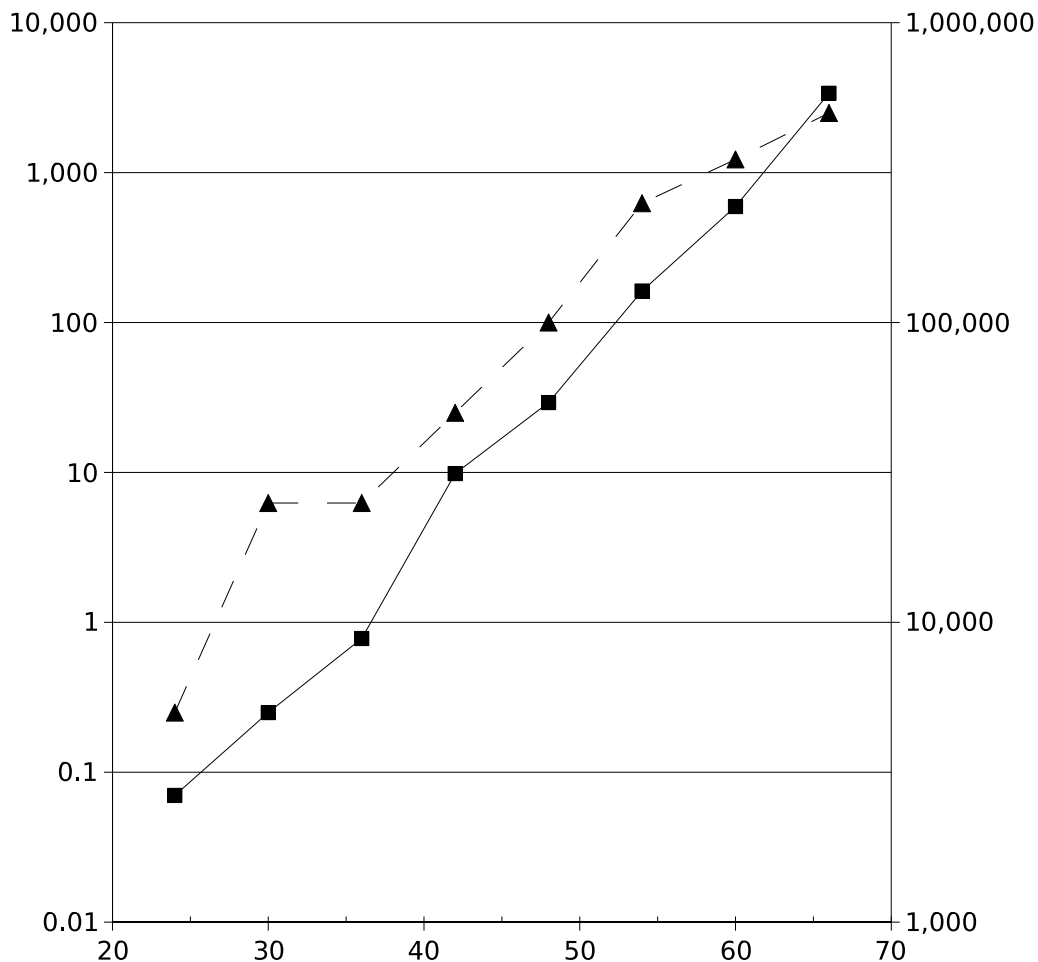


Figure 7: Diagram of the First *Number Size Comparison* Test Runs.

As predicted, the running time grows exponentially with the size of the number to be factored. There is no point in comparing the running times with those of the article. The order of magnitude is entirely another.

The optimal sieving bound according to Silverman increases exponentially, too. However, as we will see in a later section, the observed optimal sieving bounds for this system are higher than above. It is likely that this is due to the division of the sieving interval into blocks and of course, the difference in computer architecture.



The interesting issue is: Can we get faster runs with a larger sieving bound?

We can answer this question by doing test runs on the same numbers as above with the same parameters and only adjust the sieving bound. The first two numbers are skipped. Seeing the variation in order of magnitude of the running time for the numbers, they must be grouped and put into three diagrams instead of one.

The results of test runs for the 36-digit and 42-digit numbers are as follows:

<b>M</b>	<b>Running Time, 36-digit number</b>	<b>Running Time, 42-digit number</b>
25,000	0.78	
50,000	0.73	9.83
75,000	0.68	9.13
100,000	0.67	8.42
125,000	0.72	8.13
150,000	0.7	8.02
175,000	0.65	8.07
200,000	0.67	8.18
225,000	0.67	8.03
250,000	0.68	8.25
275,000	0.7	7.92
300,000	0.7	7.68
325,000	0.72	7.98
350,000	0.72	8.1

Table 5: Results of the Second/Third *Number Size Comparison* Test Runs.

The block size is 50,000 here.

Put into a diagram, it looks this way: The x-axis shows  $M$ , the primary y-axis shows the curve for the running time for the 36-digit number (solid line) and the secondary y-axis shows the curve for the running time for the 42-digit number (dashed line).

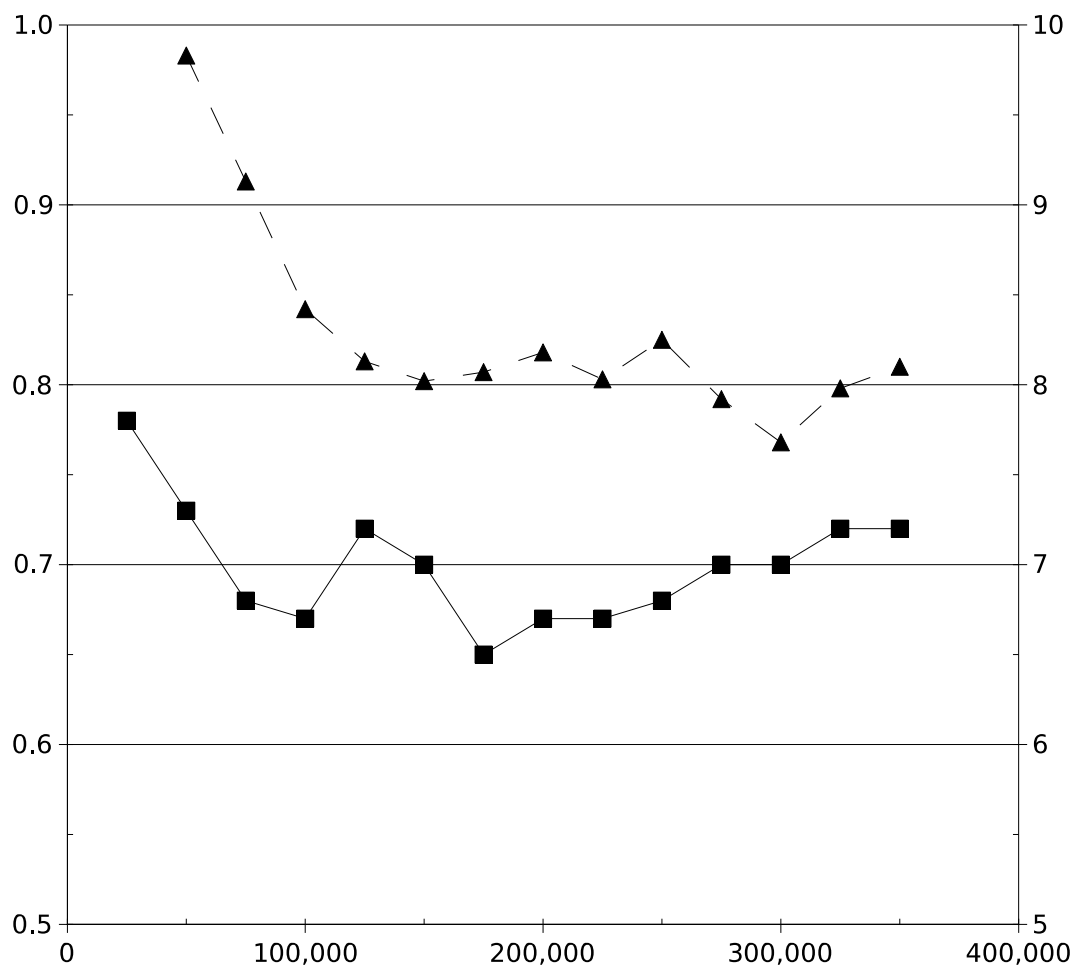


Figure 8: Diagram of the Second/Third *Number Size Comparison* Test Runs.

The results of test runs for the 48-digit and 54-digit numbers are as follows:

M	Running Time, 48-digit number	Running Time, 54-digit number
100,000	29.32	
250,000	23.23	161.85
500,000	22.72	155.3
600,000	22.48	154.52
700,000	22.1	152.42
800,000	22.53	153.2
900,000	23.85	148.1

Table 6: Results of the Fourth/Fifth *Number Size Comparison* Test Runs.

M	Running Time, 48-digit number	Running Time, 54-digit number
1,000,000	23.3	147.32
1,100,000	23.12	150.48
1,200,000	23.28	151.48
1,250,000	22.12	154.77
1,500,000	23.78	153.03
1,750,000	23.63	157.33
2,000,000	24.32	154.67

Table 6: Results of the Fourth/Fifth *Number Size Comparison* Test Runs.

The block size is now 100,000.

The diagram is analogous to the one above.

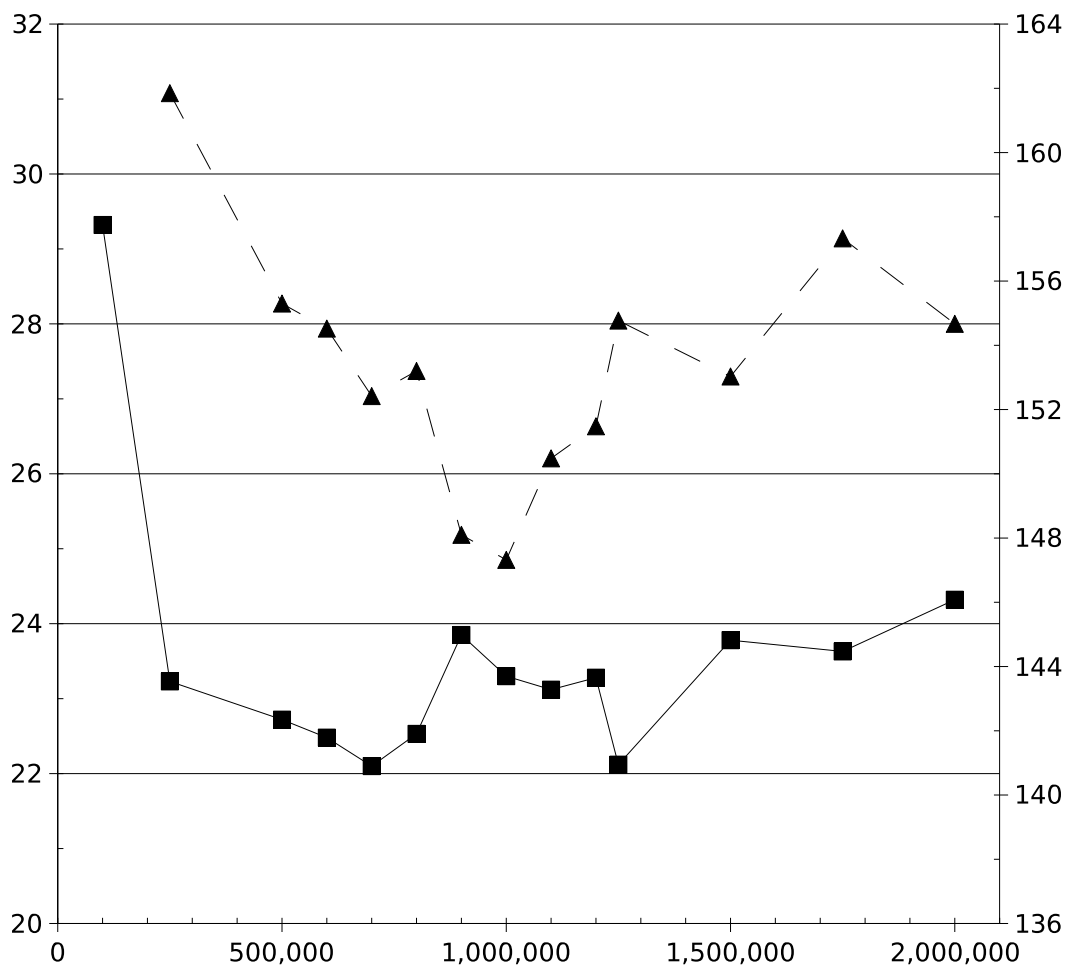


Figure 9: Diagram of the Fourth/Fifth *Number Size Comparison* Test Runs.

The results of test runs for the 60-digit and 66-digit numbers are as follows:

<b>M</b>	<b>Running Time, 60-digit number</b>	<b>Running Time, 66-digit number</b>
350,000	593.22	
500,000	565.4	3377.37
750,000	558.62	3201.9
1,000,000	546.83	3137.87
1,250,000	554.93	3118.83
1,500,000	541.62	3047.83
1,750,000	559.03	3060.18
2,000,000	561.45	3126.8
2,250,000	563.37	3145.85
2,500,000	554.73	3139.82
3,000,000	575.7	3118.93
3,500,000	567.42	3183.87

Table 7: Results of the Sixth/Seventh *Number Size Comparison* Test Runs.

The block size is still 100,000 although (as we will see later) a block size of 200,000 should give slightly faster runs. The reason for having a block size of 100,000 here is that we do not want to generate blocks where only part of the block is sieved.

The running times are starting to shoot into the sky. Whereas for the smaller numbers, one could wait for the factoring algorithm to finish, we now have running times of 9-10 minutes respectively 50-60 minutes. Here, it really pays to try and minimise the running time by raising the sieving bound. Also, as will be discovered later, a faster computer will give significantly faster runs.

The diagram looks similar to the ones above and we can conclude that this also would be the case for larger numbers. The x-axis shows  $M$ , the primary y-axis shows the curve for the running time for the 60-digit number (solid line) and the secondary y-axis shows the curve for the running time for the 66-digit number (dashed line).

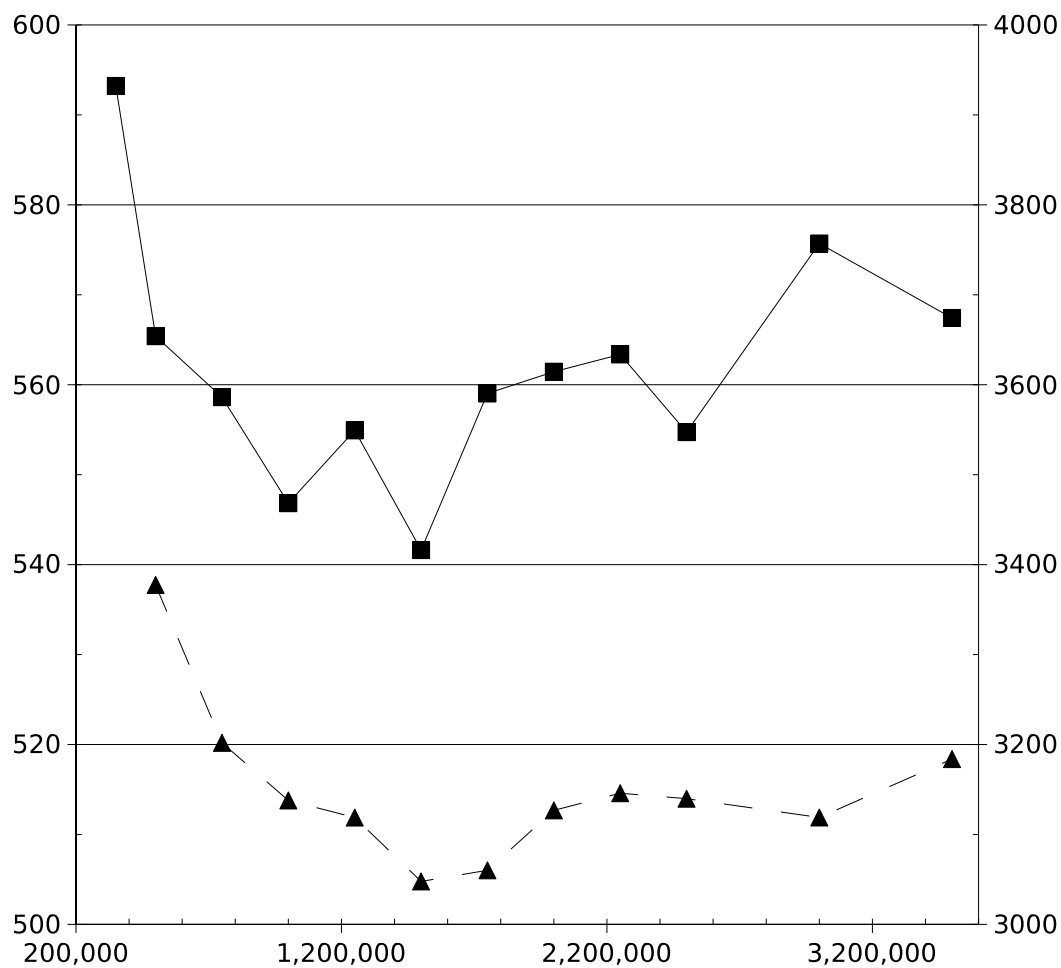


Figure 10: Diagram of the Sixth/Seventh *Number Size Comparison* Test Runs.

All of the curves seem to be more or less W-shaped but they never reach as high as they start. In other words, we can only gain by choosing a somewhat higher sieving bound than suggested by Silverman. There are several local minima in each curve. However, the proposedly optimal values can clearly be distinguished.

**Conclusion:** The optimal sieving bounds on this system for the given numbers with between 36 and 66 digits are displayed below in table 8.

**Note:** The values in the table are approximate and apply specifically to the numbers used in the test runs. To be able to make a more general statement, one would have to choose more numbers of the same sizes and repeat the test runs for those.

Number of Digits	Sieving Bound
36	175,000
42	300,000
48	750,000
54	1,000,000
60	1,500,000
66	1,500,000

Table 8: Optimal Sieving Bounds.

Seeing that it took less than an hour factoring the biggest number, we can extend the table given in [29] and factor larger numbers as well.

The results of test runs are as follows:

Size of the Number	M	Factor Base Size	Large Prime Bound	Running Time (Seconds)
60 digits	1,500,000	4849	250,000,000	294.87
66 digits	2,000,000	9133	1,000,000,000	1628.35
70 digits	5,000,000	13113	100,000,000	3530.9
80 digits	10,000,000	20815	100,000,000	33916.22

Table 9: Results of the Eighth *Number Size Comparison* Test Runs.

According to these numbers, it would take three days to factor a 90-digit number and a month to factor a 100-digit number.

Note: Parameters that are more suited for this system were used here, that is why the test runs say that the system is almost as fast for 70-digit numbers as for the 66-digit number of the first test run according to table 4 above.

Put into a diagram, it looks this way: The x-axis shows the size of the number, the primary y-axis shows the curve for the running time (solid line) and the secondary y-axis shows the curve for the sieving bound  $M$  (dashed line). Both y-axes have a logarithmic scale.

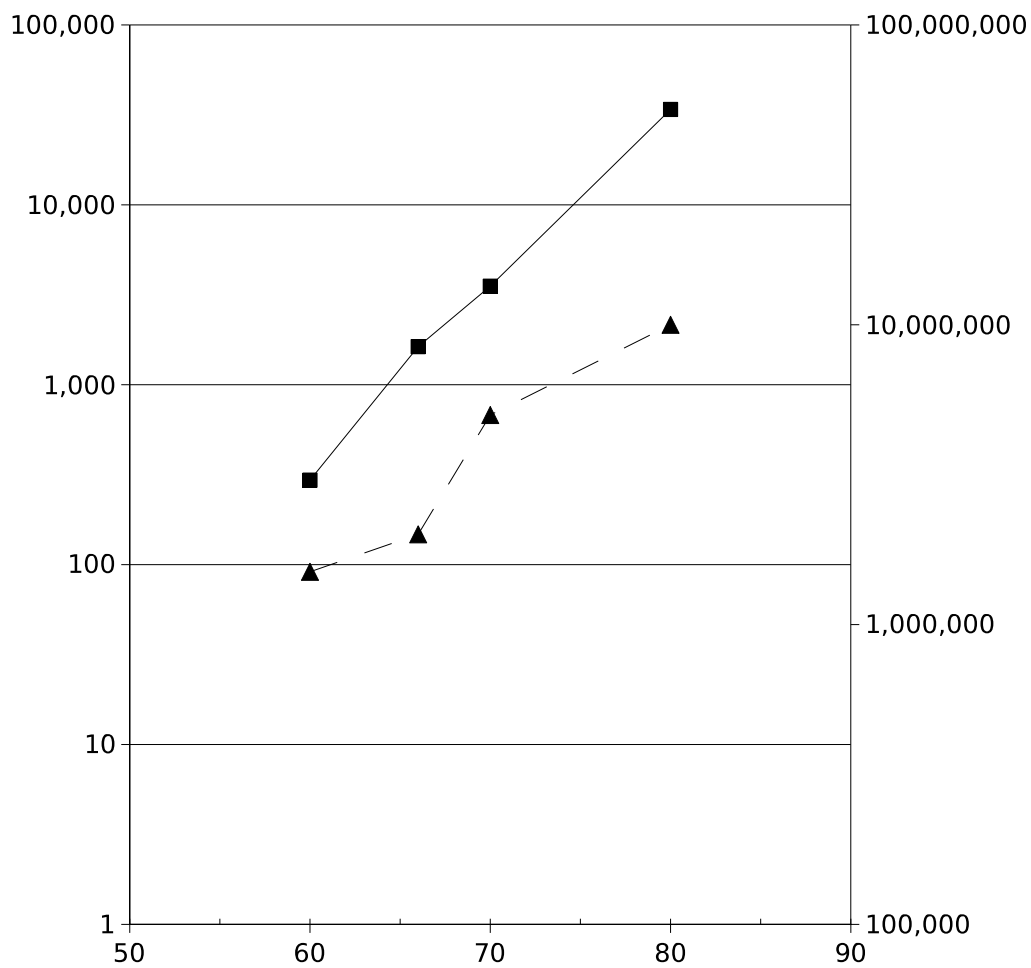


Figure 11: Diagram of the Eighth *Number Size Comparison* Test Runs.

Like before, the running time increases exponentially with growing number size.

Comparison test runs on the Linux/Fedora system give the following results:

Size of the Number	M	Factor Base Size	Large Prime Bound	Running Time (Seconds)
60 digits	1,500,000	4849	250,000,000	201.65
66 digits	1,500,000	9133	100,000,000	982.17
70 digits	5,000,000	13113	100,000,000	2410.83
80 digits	10,000,000	20815	100,000,000	22709.67

Table 10: Results of the Ninth *Number Size Comparison* Test Runs.

Put into a diagram, it looks this way: The x-axis shows the size of the number, the y-axis shows the running times for the two systems as dark grey bars for the Linux cluster/Red Hat system and light grey bars for the Linux/Fedora system.

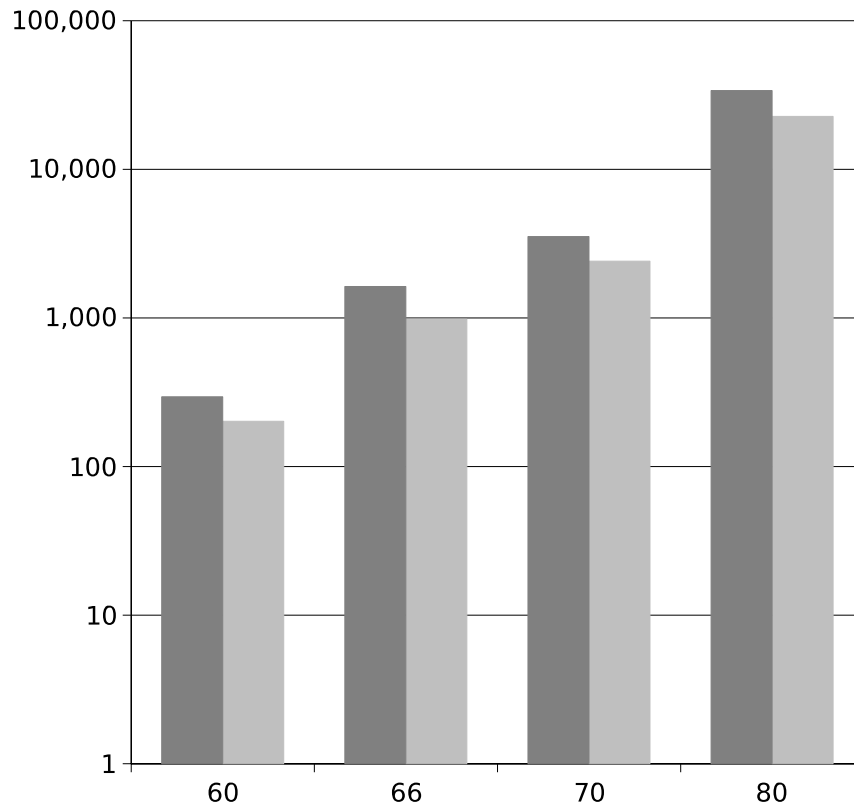


Figure 12: Diagram of the Ninth *Number Size Comparison* Test Runs.

The Linux/Fedora system is a bit faster than the Linux cluster/Red Hat system. Other than that, the diagram agrees well with our previous observations and also with the observations that will be made in section 5.6.

As we go beyond 80 digits, memory usage becomes an issue. For a 90-digit number, the available 2 GB on the Linux/Fedora system were not sufficient to complete the test run. This is due to the fact that partial relations are kept in main memory and merged whenever a pair with the same large prime is found. A solution to the problem is to dump all partial relations into a file and merge afterwards. We would then have to estimate the needed number of partial relations beforehand.



## 5.2 Block Size Comparison

The block size is the size of one block which is sieved in one piece and  $M$  is the upper bound of the overall sieving interval (see section 4.4). This means that a value of  $M = 1,400,000$  and a block size of  $bs = 700,000$  gives four sieving runs per polynomial (since the sieving interval ranges from  $-1,400,000$  to  $+1,400,000$ ).

So what happens when you increase/decrease the block size?

At first sight, a larger block size would be preferable, since that means we get fewer blocks when sieving and thus less overhead when switching blocks.

The number of polynomials generated should remain the same regardless of block size, because all we do is splitting the sieving interval into more or less blocks.

Furthermore, there should not be any influence on other parts of the algorithm than the sieving, because we do not change other parameters and the sieving is virtually independent from the rest of the program.

The results of the test runs are as follows:

Size of the Number	M	Factor Base Size	Smooths Required
60 digits	1,400,000	3035	3040

Table 11: Parameters of the First *Block Size Comparison* Test Runs.

Block Size	Running Time (Seconds)	Sieving Time (Seconds)	Number of Polynomials
56,000	377.38	257.77	2501
80,000	348.03	225.13	2507
112,000	331.45	206.85	2511
140,000	320.37	196.8	2515
175,000	312.25	191.17	2510
200,000	307.02	185.3	2511
280,000	319.42	193.38	2539

Table 12: Results of the First *Block Size Comparison* Test Runs.

Block Size	Running Time (Seconds)	Sieving Time (Seconds)	Number of Polynomials
400,000	418.62	294.82	2528
560,000	899.05	773.02	2587
700,000	1079.62	952.5	2566
1,400,000	1769.15	1641.15	2623

Table 12: Results of the First *Block Size Comparison* Test Runs.

Put into a diagram, it looks this way: The x-axis shows the block size, the primary y-axis shows the curves for the running time (solid line) and the sieving time (dashed line) and the secondary y-axis shows the curve for the number of polynomials generated (dashed-dotted line).

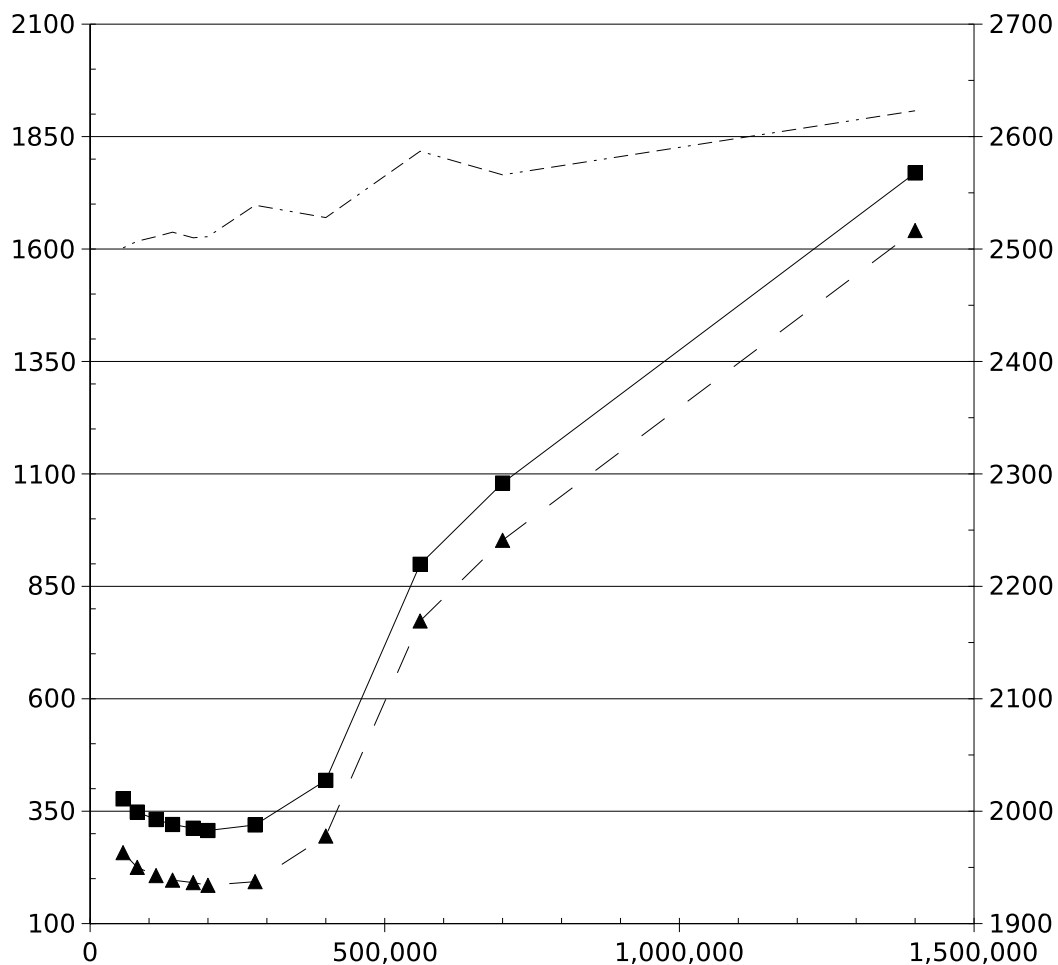


Figure 13: Diagram of the First *Block Size Comparison* Test Runs.

The results in table 12 and figure 13 paint a different picture than expected. What is the reason? The answer is that the sieving time is highly governed by memory access times. As Scott Contini suggests in his thesis (see [27]), the sieving should become much faster when the entire sieving block fits into the cache of the processor and need not be swapped in and out of the processor cache all the time. Here, it seems that blocks with a block size below about 300,000 fit into the cache and therefore are sieved faster. As the block size increases, the sieving block needs to be stored in RAM and part of the block is swapped into the processor cache when needed. The larger the sieving block, the larger the amount of memory that is accessed frequently and swapped in and out of the cache and the sieving takes more and more time. As for the sieving time for small blocks that fit into the cache, it decreases with growing block size which is probably due to overhead as mentioned above. However, the differences are small and different test runs with the same parameters may vary in time with a small percentage anyway.

What about the number of polynomials generated? Local fluctuations can be seen, but all in all it seems that the number of polynomials generated increases linearly with increasing block size. This is not surprising, as the whole sieving block is initialised with the same sieve location (see section 4.1). Hence larger blocks tend to get insensitive to smaller function values in the block and do not detect as many probably smooth function values. Note: The  $M$  in the initial sieve location formula is replaced by the largest value in the sieving block, so that the sieve location becomes more suitable to the current block. In other words, this is not so much a given disadvantage for larger blocks, rather an optimisation for smaller blocks. Otherwise, all blocks regardless of size would be initialised with the same sieve location, fewer smooth values would be found for the inner blocks and more polynomials would be necessary.

It can also be seen that a deviation from the curve leads to a corresponding deviation in sieving time resp. running time. Example:  $bs = 560,000$ . Naturally, when more polynomials are generated, more blocks are sieved and the running time is longer.

As expected, the diagram does not indicate any influence on other parts of the program than the sieving.

To be on the safe side, we need something to compare with so that we can undermine the claim about the cache size.

How does the same curve look for a different type of processor?

Since the cache of the Linux/Red Hat system is only half as big as the one of the Linux/Mandrake system, there should be a difference, but the shape of the curve should remain the same.

The results of the test runs on the other system are as follows:

<b>Block Size</b>	<b>Running Time (Seconds)</b>	<b>Sieving Time (Seconds)</b>	<b>Number of Polynomials</b>
28,000	589.4	437.43	2495
40,000	517.12	363.5	2496
56,000	469.45	314.52	2501
70,000	451.52	296.67	2506
80,000	451.83	297.67	2507
100,000	452.63	298.93	2508
140,000	456.32	302.07	2515
175,000	458.82	301.55	2510
200,000	466.8	311.87	2511
280,000	628.22	469.48	2539
350,000	882.6	723.48	2510
400,000	1080.72	919.53	2528
560,000	1413.17	1248.6	2587

Table 13: Results of the Second *Block Size Comparison* Test Runs.

The other parameters are the same as above.

We can see at once that the running times are slower even for small block sizes, because the Linux/Red Hat system is not as powerful as the Linux/Mandrake system (see section 3.4).

The number of polynomials generated stays the same, considering that exactly the same calculations are performed.

Put into a diagram, it looks this way: The x-axis shows the block size, the primary y-axis shows the curves for the running time (solid line) and the sieving time (dashed line) and the secondary y-axis shows the curve for the number of polynomials generated (dashed-dotted line).

Note that the scale of the x-axis is different from the one in the above diagram.

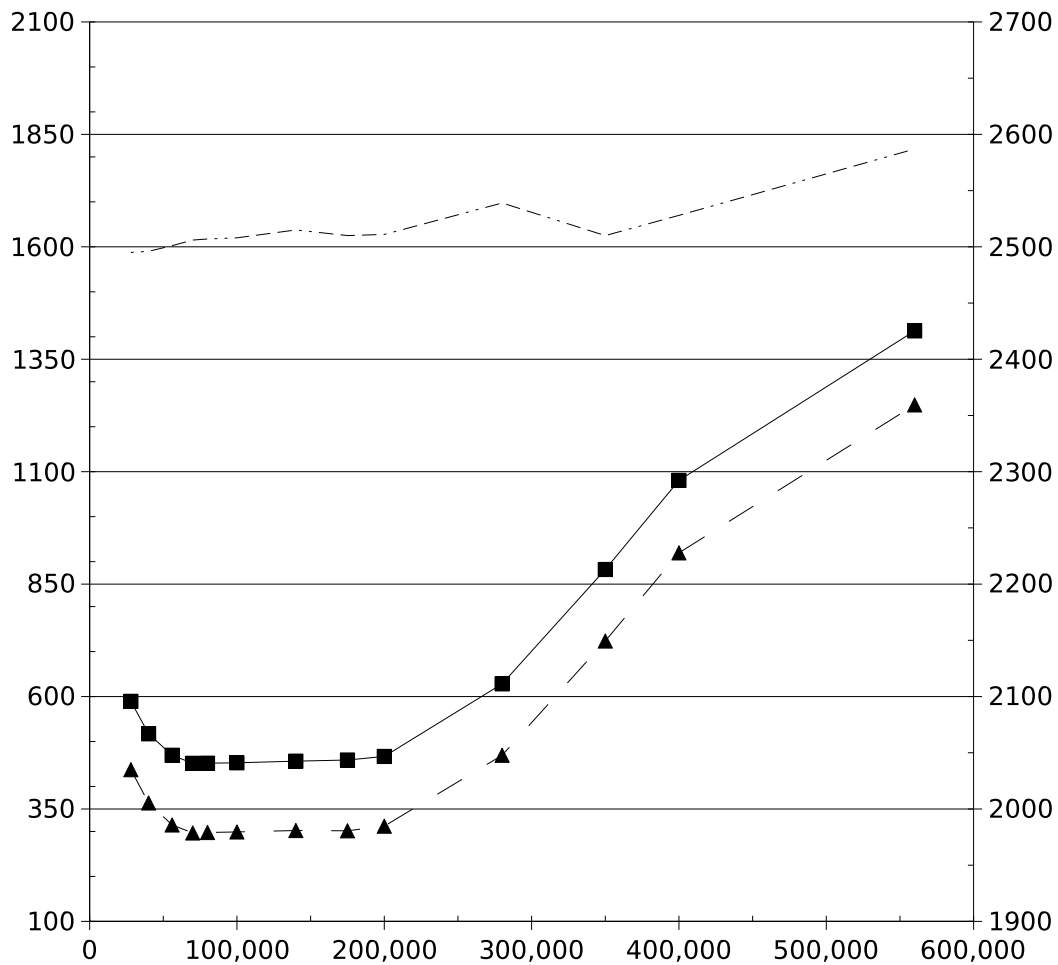


Figure 14: Diagram of the Second *Block Size Comparison* Test Runs.

The curve is not as round at the bottom as the previous one thus lacking a clear minimum. This must be due to the processor type. Also, we would expect the curve to go up sooner, at about 150,000 instead of at 200,000. Obviously, the AMD Athlon processor cache is used more effectively for block sizes that almost fill up the whole cache. Anyway, we can clearly see that the second curve at 300,000 is comparable to the first curve at 400,000 and we conclude that the assumption about the cache was right. Once we have a block size that exceeds the limits of the processor cache, we get very much higher running times.

**Conclusion:** Pick a block size that makes the sieving block fit into the processor cache. That increases the speed significantly. If in doubt, prefer a smaller block size, because the loss is negligible compared to picking a block size that is much too large.

### 5.3 Sieving Bound Comparison

The sieving bound  $M$  determines the size of the sieving interval which is  $2M + 1$  (again, see section 4.1).

What can one expect to see when changing the sieving bound?

If the block size is the same, a slightly larger  $M$  means that the algorithm produces more blocks per polynomial and fewer polynomials. At first sight, there should not be any big difference. But once we reach a certain value for  $M$ , the probability of finding smooth function values should drop significantly for the outer blocks and sieving should take more time.

Naturally, the number of polynomials should decrease with growing  $M$ , since we still search for the same amount of smooths. We can expect that a doubling of  $M$  leads to a halving in the number of polynomials as long as the number of smooth values per block remains approximately the same.

The results of the test runs are as follows:

Size of the Number	Block Size	Factor Base Size	Smooths Required
60 digits	140,000	3035	3035

Table 14: Parameters of the First *Sieving Bound Comparison* Test Runs.

M	Total Time	Sieving Time	Polynomial Generation Time	Factor Base Update Time	Number of Polynomials
350,000	446	156.37	75.68	193.07	7164
700,000	353.95	179.53	41.67	109.37	4083
1,050,000	351.83	207.87	31.63	85.77	3142
1,400,000	340.37	220.15	26.1	66.98	2512
1,750,000	332.68	226.72	20.77	56.35	2087
2,100,000	353.17	250.23	20.2	51.7	1889
2,800,000	358.5	269.75	15.42	40.87	1533

Table 15: Results of the First *Sieving Bound Comparison* Test Runs.

As before, all times are in seconds.

Put into a diagram, it looks this way: The x-axis shows  $M$ , the primary y-axis shows the curves for the running time (solid line), the sieving time (dotted line), the polynomial generation time (dashed-dotted line) and the factor base update time (dotted-dashed line) and the secondary y-axis shows the curve for the number of polynomials generated (dashed line).

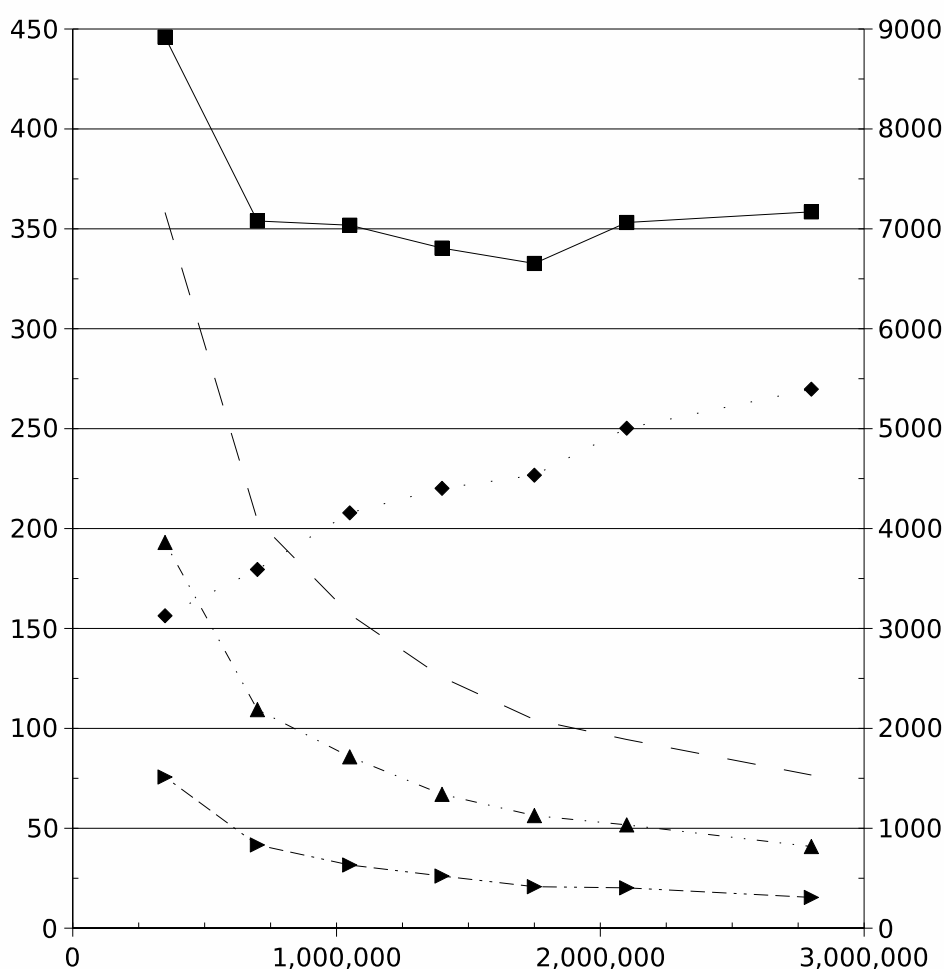


Figure 15: Diagram of the First *Sieving Bound Comparison* Test Runs.

We can see in table 14 and figure 15, that the total running time is not nearly as constant as expected. The diagram shows that it generally takes more time generating new polynomials and updating the factor base than sieving the corresponding extra amount per polynomial. But from a certain value of  $M$  on (in this case about 1,750,000), the sieving time dominates the total time to such an extent that increasing  $M$  increases instead of decreases the total time. This is further illustrated in figure 16.

The number of polynomials generated drops very fast at first and the time needed to generate them and update the factor base seems to drop equally fast. This is as predicted.

The following pie chart diagrams show the times from table 15 (and some additional times) in relation to the total running time. The white section represents the polynomial generation time, the light grey section represents the factor base update time, the grey section represents the sieving time, the dark grey section represents the trial division time and the black section represents the rest of the time.

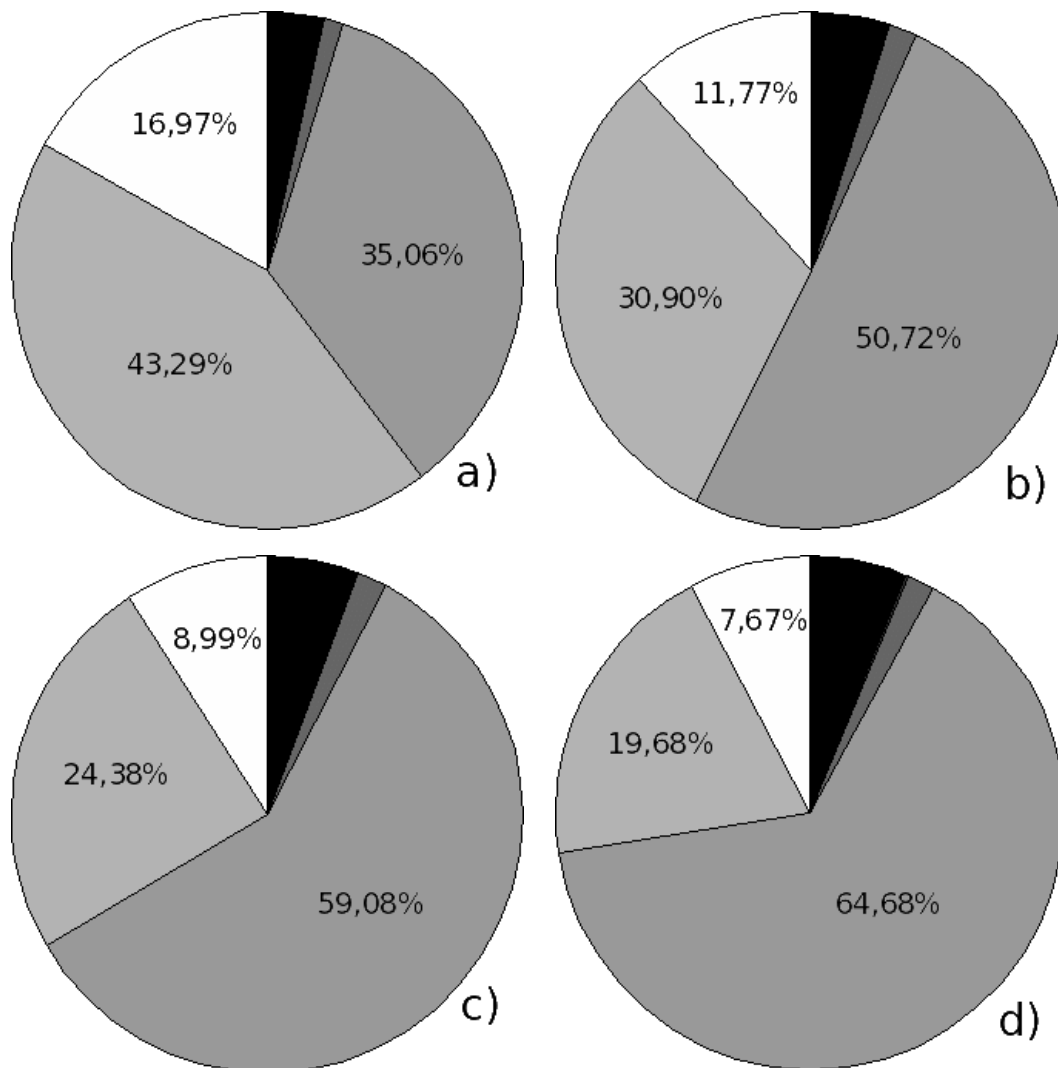


Figure 16: Pie Charts of the First *Sieving Bound Comparison* Test Runs.



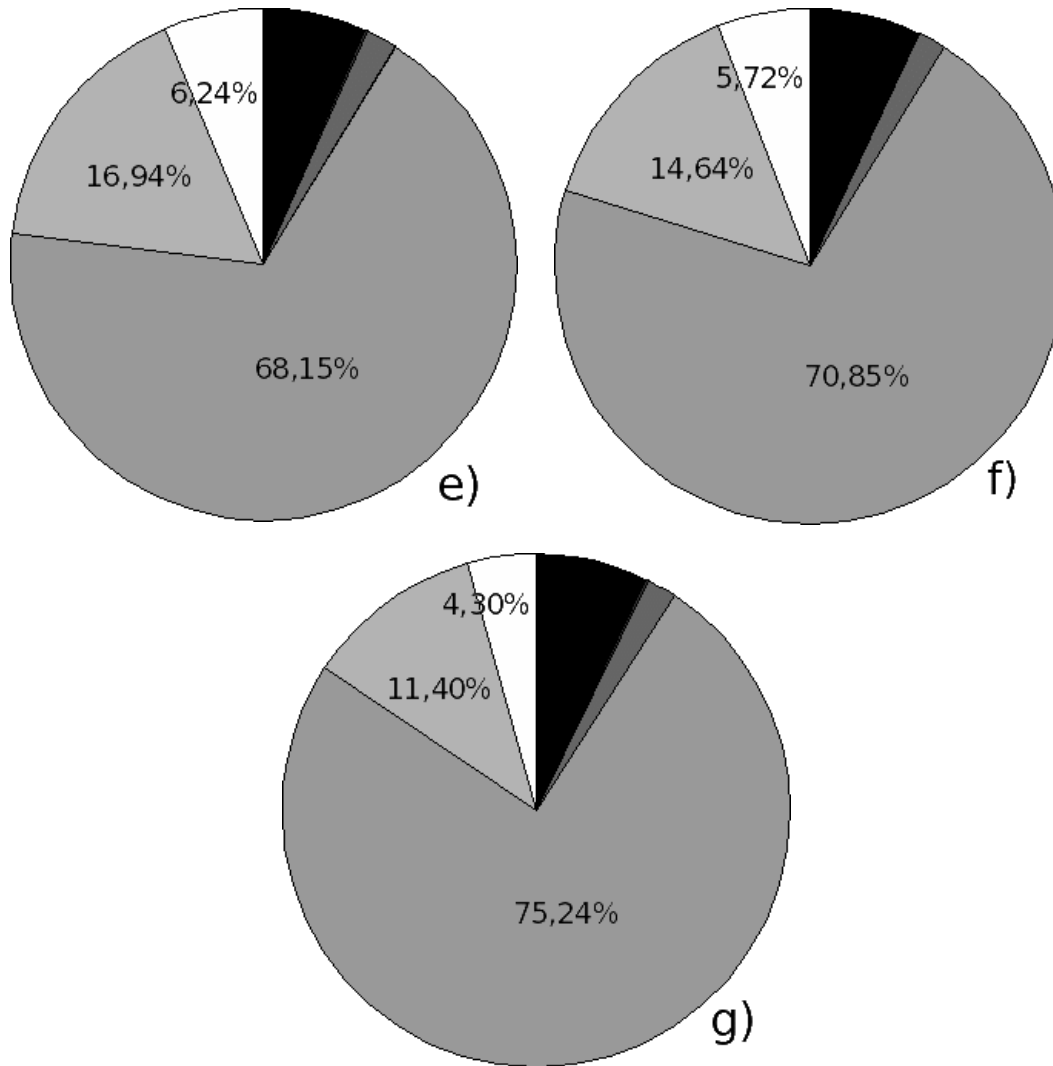


Figure 16: Pie Charts of the First *Sieving Bound Comparison* Test Runs.

Clearly, the larger the sieving bound, the bigger the part of the time that is spent on sieving, just as we have seen before. As sieving is the most vital part of the algorithm, it never takes less than  $1/3$  of the total time in these test runs. It can be noted, however, that in the first test run the factor base update takes more time than the sieving.

The supposedly optimal value for the sieving bound (1,750,000) is reached in pie chart e). It seems, that the sieving should take about  $2/3$  of the total time. The parts closely related to sieving, polynomial generation and factor base update, then take about  $1/4$  of the total time. It can also be said that those three parts together stand for more than  $9/10$  of the running time and the trial division, matrix elimination and other time are almost not noticeable until we reach the last test run. Furthermore, those times remain more or less constant over the test runs.

What happens if we increase the sieving bound further?  
 To answer that question, I performed a second series of test runs.

As mentioned at the beginning of the chapter, the probability of finding smooth values should drop rapidly for the outer blocks when  $M$  becomes very large. This is due to the fact that we get larger function values at the edges of the sieved interval. The sieving time should explode, since we sieve the outer blocks in vain.

The results of the test runs are as follows:

Size of the Number	Block Size	Factor Base Size	Smooths Required
60 digits	200,000	3035	3040

Table 16: Parameters of the Second *Sieving Bound Comparison* Test Runs.

Observe that these parameters differ from the ones in the first series of test runs. Therefore, we cannot draw any conclusions from the running times of the second series compared to those of the first series of test runs. Actually, what happens between  $M = 2,800,000$  and  $M = 4,200,000$  is obvious: The running time increases and the number of polynomials generated decreases.

Note: For the last four values below, the outer blocks do not produce any smooth values. The last block to contain smooth values is block 256 and the last four runs have 266, 322, 378 resp. 434 blocks (that is  $M$  divided with half the block size, 100,000).

M	Total Time	Sieving Time	Number of Polynomials
4,200,000	332.42	252.38	1152
9,800,000	386.45	316.67	624
15,400,000	449.88	379.08	475
21,000,000	474.03	405.18	372
26,600,000	498.82	428.65	311
32,200,000	531.68	459.62	276
37,800,000	534.38	464.1	237
43,400,000	569.73	496.6	222

Table 17: Results of the Second *Sieving Bound Comparison* Test Runs.

Put into a diagram, it looks this way: The x-axis shows  $M$ , the primary y-axis shows the curves for the running time (solid line) and the sieving time (dotted line) and the secondary y-axis shows the curve for the number of polynomials generated (dashed line).

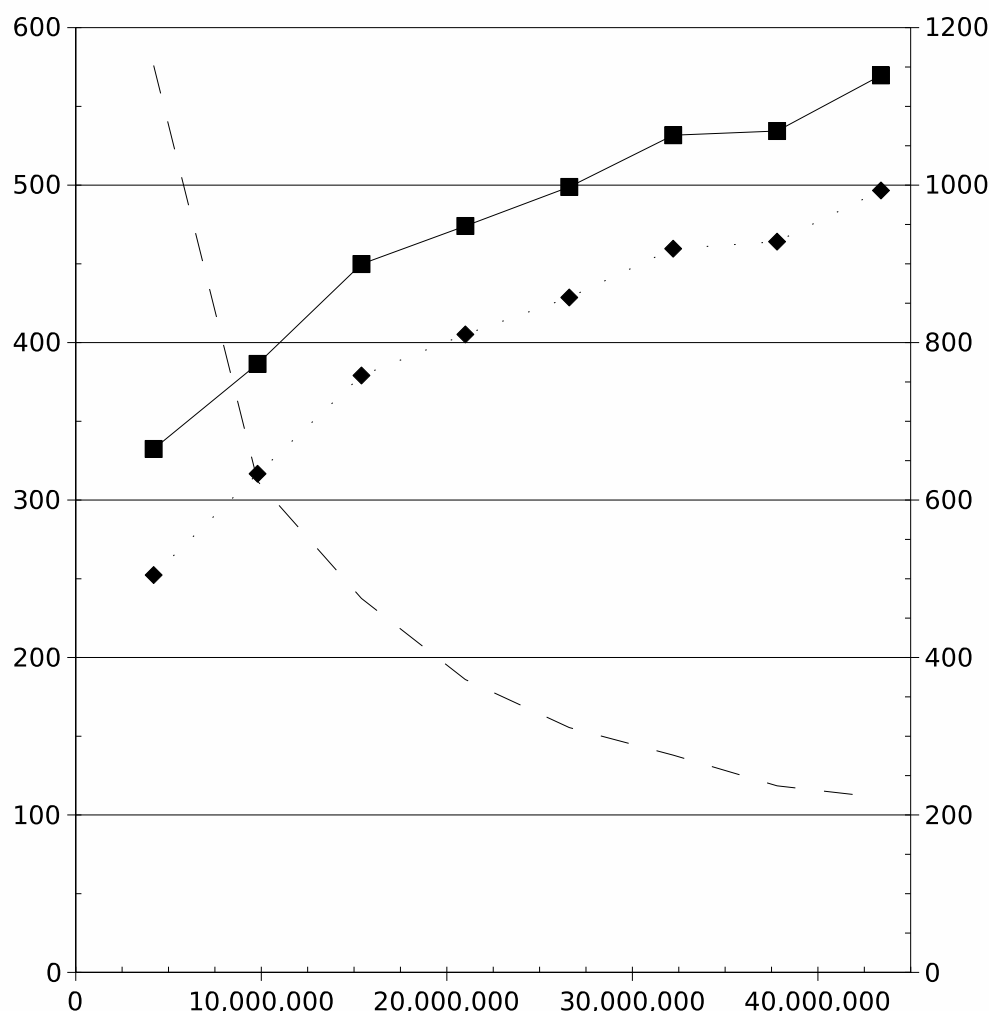


Figure 17: Diagram of the Second *Sieving Bound Comparison* Test Runs.

The slope of the curve is not nearly as steep as expected. Why? For one thing, the “original” curve above at ca. 3,000,000 is very flat to begin with. We can still see some change in the curve between 21,000,000 and 26,600,000 where it becomes almost linear instead of logarithmic as before. For the other, the number of polynomials still abates, although no new smooths are found for the outer blocks once the sieving bound passes 25,500,000. That must mean that we find useful semi-smooth values in the outer blocks. With  $M > 40,000,000$ , the probability for semi-smooth values obviously drops, too.

It seems, the optimal value of the sieving bound is 1,750,000. This is of course strongly related to the size of the number. The question is: What do the curves for other 60-digit numbers look like?

The results of the test runs are as follows:

Size of the Number	Block Size	Smooths Required	System
60 digits	100,000	factor base size + 5	Linux cluster/ Red Hat

Table 18: Parameters of the Third *Sieving Bound Comparison* Test Runs.

The first 60-digit number is the same as previously, its first digit is 1. The second number begins with 2, the third with 4, the fourth with 6, the fifth with 8 and the sixth with 9. Note that every number has a different factor base size, since I chose the parameter to be the upper limit of the primes contained in the factor base.

The total times are:

M	No. 1	No. 2	No. 3	No. 4	No. 5	No. 6
250,000	569.52	857.18	845.02	1385.78	994.37	708.37
500,000	449.75	686.65	678.62	1088.68	772.13	552.58
750,000	411.9	614.4	625.38	970.07	706.47	516.58
1,000,000	386.68	577.53	601.33	941.85	684.67	495.53
1,250,000	386.58	588.92	611.25	936.23	681	491.72
1,500,000	377.23	599.22	595.45	936.47	684.35	488.32
1,750,000	379.22	600.75	597.42	939.88	674.47	500.9
2,000,000	396.03	603.9	604.53	932.23	674.02	491.15
2,250,000	405.67	604.17	611.97	925.62	686.73	502.2
2,500,000	405.28	610.8	611.62	953.93	667.7	514.62

Table 19: Results of the Third *Sieving Bound Comparison* Test Runs.

Put into a diagram, it looks this way: The x-axis shows  $M$  and the y-axis shows the curves for the running time. A look at table 19 above tells which curve belongs to which number.

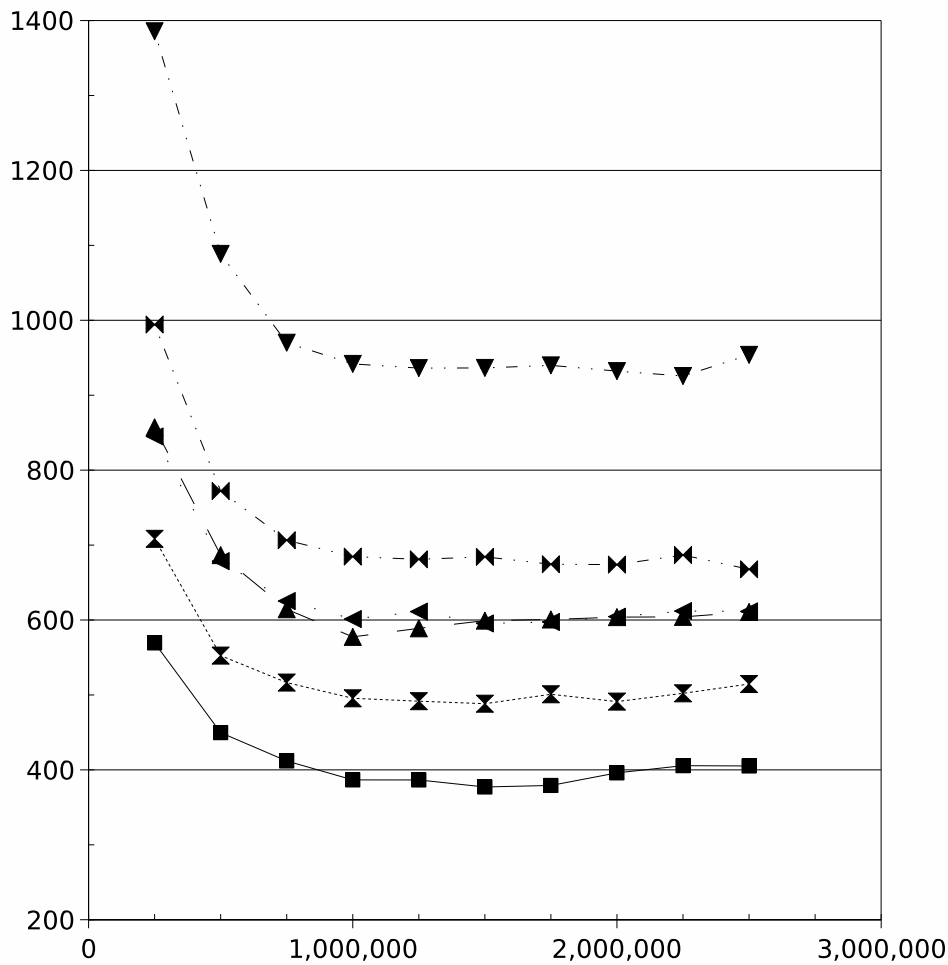


Figure 18: Diagram of the Third *Sieving Bound Comparison* Test Runs.

Fortunately, the curves look very much alike. The surprising thing is that the curves are not “in order”, i.e. it is not the case that the biggest number takes the longest time to factor. That does not depend on the factor base sizes though, which are 3035, 3066, 3076, 3036, 3038 and 3046 for the six numbers respectively. Apparently, some numbers are more difficult to sieve than others.

Conclusion: For 60-digit numbers, choose  $M$  between 1,000,000 and 2,500,000. With an adequate sieving bound, the running time can be kept low. As the diagrams show, if we choose a tiny sieving bound, or a very large sieving bound, the time almost doubles which means several minutes in this case. However, it is not vital to hit the exact minimum of the curve and we have some leeway. The best thing to do for a new number is to sieve a small part so that we get a certain fraction of the desired amount of smooths, and adjust the sieving bound until having found a good value.

## 5.4 Large Prime Bound Comparison

The large prime bound governs how large proportion of the total number of valuable relations consists of semi-smooth relations (see section 4.1). Naturally, if the large prime bound is less or equal to the factor base prime bound, no partial relations are taken into consideration.

For 60-digit numbers, Silverman suggests in his article a large prime bound of  $B1^{2.4}$ ,  $B1$  being the factor base prime bound. That is approximately  $2.935 \times 10^{11}$ . He predicts that one can easily half the running time with an appropriate large prime bound.

How well does that agree with our system?

What happens for larger or smaller large prime bounds?

As we have discovered before, sieving is costly and the more relations we find per block, the less blocks we have to sieve. So we can guess that Silverman is right and that a high large prime bound is much better than no large prime bound.

Since the same blocks are sieved in each run, more relations per block would also mean fewer polynomials to generate.

On the other hand, more relations are tagged “probably smooth” and need to be checked.

The results of the test runs are as follows:

Size of the Number	Blocksize	M	Factor Base Size	Smooths Required	System
60 digits	200,000	1,600,000	3035	3040	Linux cluster/ Red Hat

Table 20: Parameters of the First *Large Prime Bound Comparison* Test Runs.

Large Prime Bound	Running Time	Sieving Time	Trial Division Time	Number of Polynomials
60,000	441.5	277.73	4.25	2825
100,000	421.35	261.53	5.6	2691
450,000	371.58	228.43	7.62	2355

Table 21: Results of the First *Large Prime Bound Comparison* Test Runs.

Large Prime Bound	Running Time	Sieving Time	Trial Division Time	Number of Polynomials
1,000,000	322.35	195.77	10.27	2033
4,500,000	284.55	168.18	13.62	1763
10,000,000	257.13	149.85	17.43	1554
25,000,000	236.93	136	20.17	1382
45,000,000	225.88	123.2	27.75	1282
100,000,000	226.1	124.25	26.62	1282
250,000,000	219.88	116.63	33.63	1199
450,000,000	220.03	110.5	40.48	1142
$1 \times 10^9$	223.47	105.15	52.7	1096
$2.5 \times 10^9$	234.32	102.27	68.68	1071
$4.5 \times 10^9$	236.38	105.03	67.67	1071
$1 \times 10^{10}$	251.58	103.13	85.15	1051
$2.5 \times 10^{10}$	272.28	99.58	109.83	1035
$4.5 \times 10^{10}$	302.18	100.17	138.88	1025
$1 \times 10^{11}$	301.62	99.15	139.75	1025
$2.935 \times 10^{11}$	341.58	100.68	178.12	1015
$4 \times 10^{11}$	394.45	97.92	233.52	1012
$8 \times 10^{11}$	394.33	97.55	233	1012
$9 \times 10^{11}$	466.23	98.4	303.08	1010

Table 21: Results of the First *Large Prime Bound Comparison* Test Runs.

Again, all times are in seconds.

Put into a diagram, it looks this way: The x-axis shows the large prime bound in logarithmic scale, the primary y-axis shows the curves for the running time (solid line), the sieving time (dotted line) and the trial division time (dotted-dashed line) and the secondary y-axis shows the curve for the number of polynomials generated (dashed line).

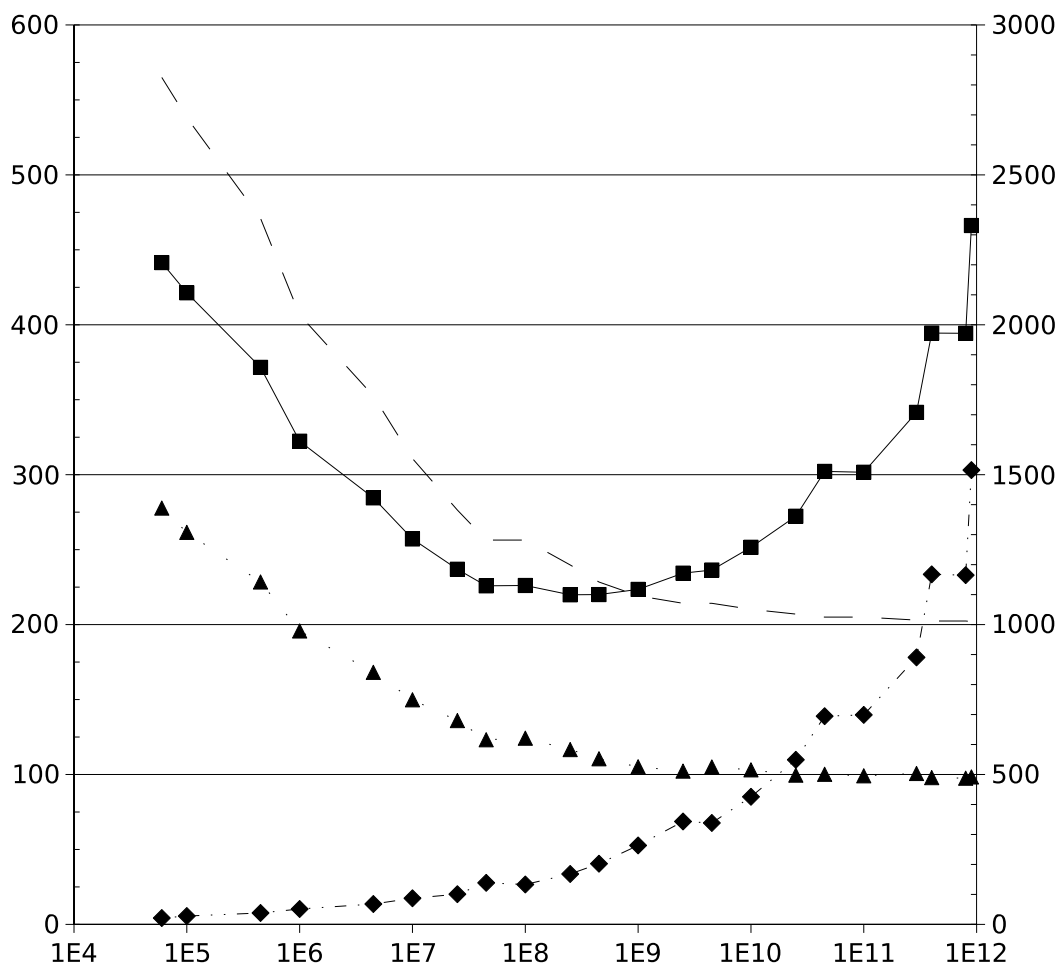


Figure 19: Diagram of the First *Large Prime Bound Comparison* Test Runs.

Looking at the running time, we find that a large prime bound between 25,000,000 and 4,500,000,000 gives the lowest values with between ca. 220 and 240 seconds. With a lower or higher large prime bound, the running time increases. Why is that? As stated before, with an appropriate large prime bound, we can find extra relations at a low cost and the consequence is that we do not need to sieve as many blocks and generate as many polynomials as without large primes. But the cost is not zero. The diagram shows that the trial division time grows very fast and at 25,000,000,000 ( $= 2.5 \times 10^{10}$ ) it even exceeds the sieving time. The sieving time in turn decreases steadily but can be approximated with 100 seconds from 1,000,000,000 on. The number of polynomials generated drops nearly at the same rate.

What about the large prime bound suggested by Silverman?

For this system,  $2.935 \times 10^{11}$  is much too high and the running time is more than 50% longer than the shortest running time measured.



Let us investigate another 60-digit number and a 70-digit number for comparison.

The results of the test runs are as follows:

Size of the Number	Blocksize	M	Factor Base Size	Smooths Required	System
60 digits	200,000	1,600,000	3035	3040	Linux cluster/ Red Hat

Table 22: Parameters of the Second *Large Prime Bound Comparison* Test Runs.

These parameters are the same as above.

Large Prime Bound	Running Time	Sieving Time	Trial Division Time	Number of Polynomials
1,000,000	471.17	290.8	13.22	2891
2,500,000	421.02	256.22	16.57	2546
10,000,000	381.7	228.25	21.6	2261
25,000,000	361.02	211.93	29.12	2095
100,000,000	350.65	199.5	34.15	1981
250,000,000	345.6	190.43	42.75	1887
450,000,000	346.52	183.6	55.27	1822
$1 \times 10^9$	355.42	180.57	68.67	1784
$2.5 \times 10^9$	370.77	177.53	88.6	1753
$1 \times 10^{10}$	392.25	174.57	114.2	1727
$2.5 \times 10^{10}$	424.32	172.77	147.22	1715
$1 \times 10^{11}$	465.75	172.47	190.17	1700

Table 23: Results of the Second *Large Prime Bound Comparison* Test Runs.

Once again, all times are in seconds.

The running times are a bit higher, since the number is larger.

The diagram below is analogous to the previous one.

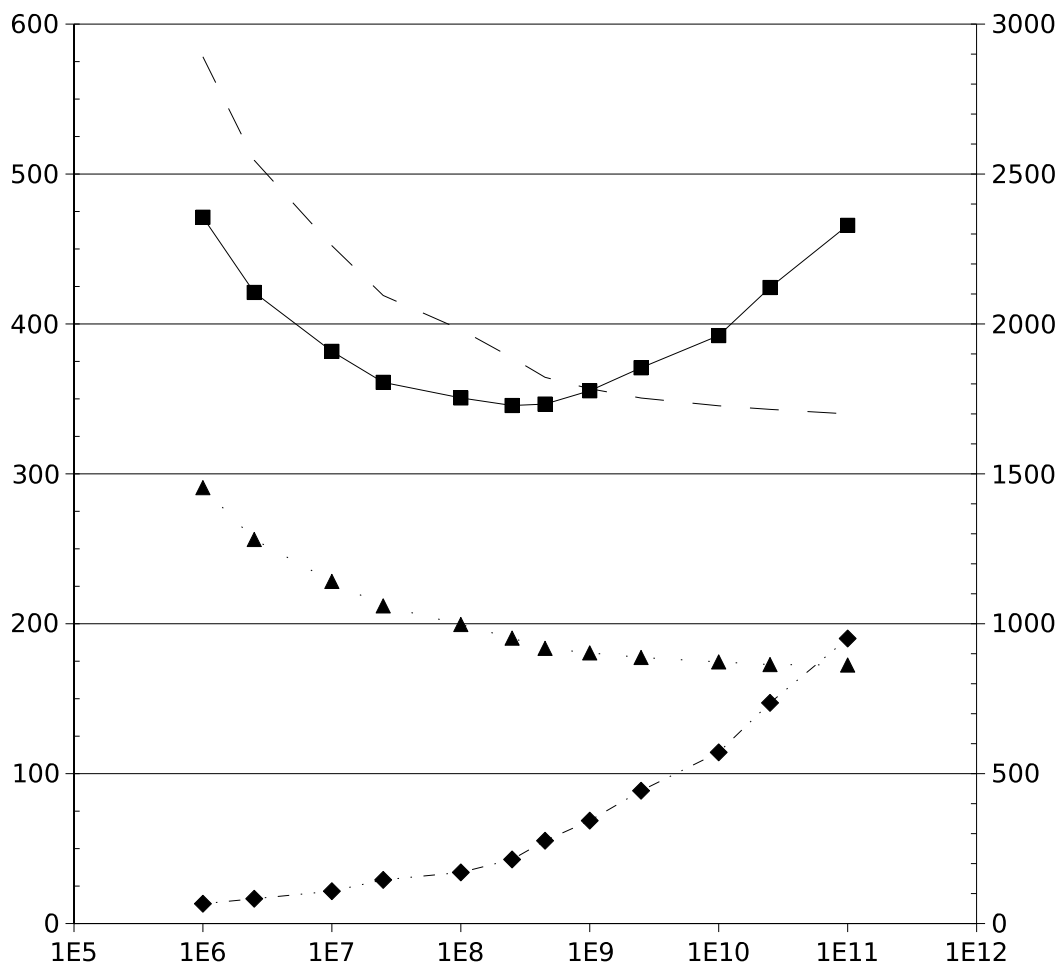


Figure 20: Diagram of the Second *Large Prime Bound Comparison* Test Runs.

These curves have the same shapes as the others above.

As pointed out, we can see that the running time curve is higher up, that is to say it takes more time to factor the number (which is about four times as large as the previous one). Also, the running time curve and the sieving time curve are farther apart which indicates that other parts of the algorithm also take more time.

It seems that the sieving time is approximately the same at the beginning of the curve, but that is due to the fact that the first curve starts with a smaller large prime bound.

A look at the numbers tells us that the total running time climbs even earlier than before, although the trial division time does not exceed the sieving time until the large prime bound is  $1 \times 10^{11}$ . This is probably because the sieving time decreases a little more slowly than before.

The minimum however can be found at exactly the same spot and

it seems that a large prime bound of 250,000,000 is a good value for 60-digit numbers on this system.

Now let us have a look at the 70-digit number.

The results of the test runs are as follows:

Size of the Number	Blocksize	M	Factor Base Size	Smooths Required	System
70 digits	200,000	5,600,000	14996	15001	Linux cluster/ Red Hat

Table 24: Parameters of the Third *Large Prime Bound Comparison* Test Runs.

Large Prime Bound	Running Time	Sieving Time	Trial Division Time	Number of Polynomials
1,000,000	4764.28	3159.67	215.12	6005
2,500,000	4069.38	2640.6	273.23	4966
10,000,000	3664.58	2306.07	351.83	4317
25,000,000	3450.27	2076.05	452.3	3940
100,000,000	3411.84	1976.73	568.68	3686
250,000,000	3425.62	1871.32	715.03	3514
450,000,000	3533.15	1827.43	898.27	3392
$1 \times 10^9$	3674.15	1741.25	1136.07	3316
$2.5 \times 10^9$	3932.27	1715.92	1433.98	3266
$1 \times 10^{10}$	4298.6	1695.42	1796.88	3230
$2.5 \times 10^{10}$	4673.57	1687.28	2213.73	3206
$1 \times 10^{11}$	5230.42	1694.63	2760.67	3192
$2.5 \times 10^{11}$	5897.77	1696.7	3434.73	3178

Table 25: Results of the Third *Large Prime Bound Comparison* Test Runs.

The diagram below is analogous to the previous ones.  
This time, the scales of the y-axes are different.

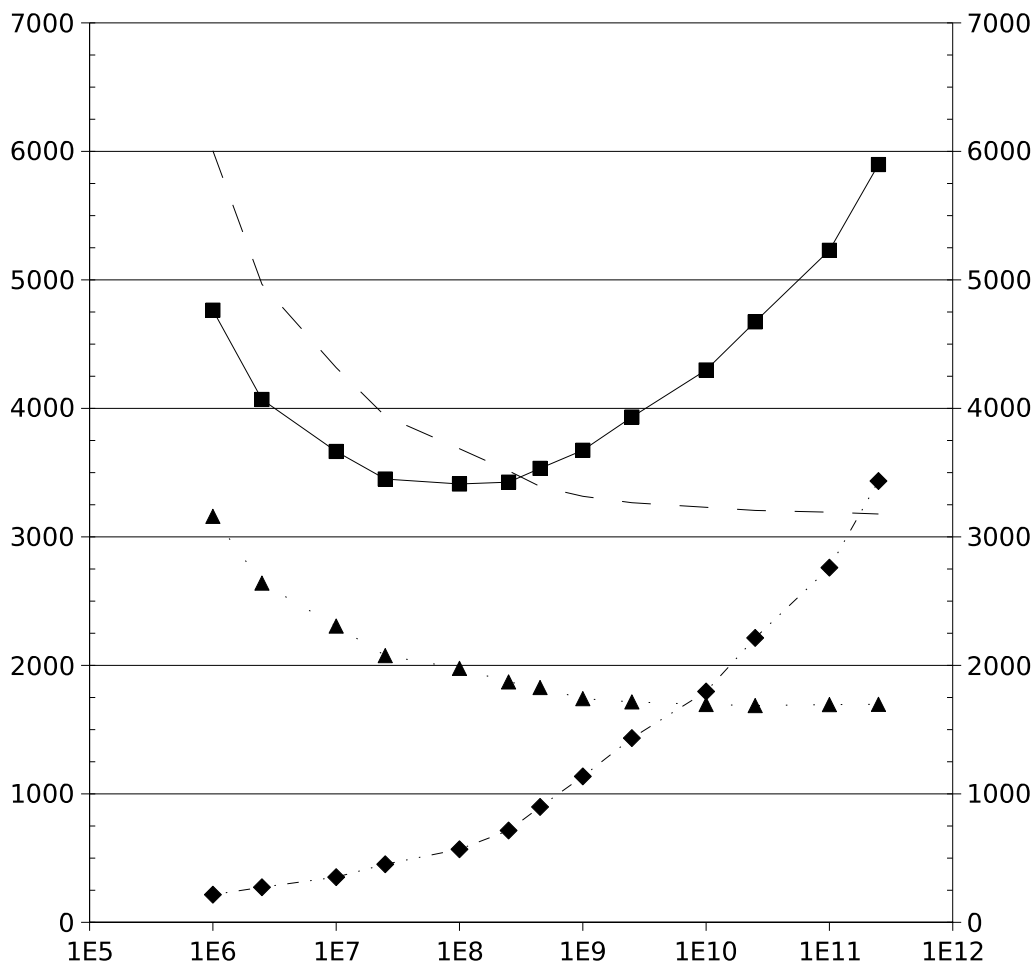


Figure 21: Diagram of the Third *Large Prime Bound Comparison* Test Runs.

All times are higher than before, of course. Apart from that, the curves look very similar.

The surprising thing is that the optimal value appears to be lower than before. One would expect the opposite effect. This observation can only be ascribed to the characteristics of this particular system.

Conclusion: The large prime bound suggested by Silverman is too high for this system.

For a 60-digit number, one should rather choose a large prime bound between  $B1^{1.55}$  and  $B1^{1.95}$  (with  $B1 = 60,000$ ).

Curiously, it looks like the optimal large prime bound for 70-digit numbers on this system is only about 100,000,000, compared to ca. 250,000,000 for 60-digit numbers.

## 5.5 Factor Base Size Comparison

The sieving relies on a proper factor base which is generated at the beginning of the algorithm (see section 2.1). The factor base sizes used so far are based on what Silverman writes in his article (see [29]).

But how well does that fit this particular system?

And what happens when we change the factor base size?

Well, a larger factor base means that we will have a higher probability of finding smooths. However, when the primes in the factor base get larger than the function values themselves, it would be of no use to further increase the size of the factor base. In practice, this case should never occur for the numbers we are trying to factor.

Note: Silverman specifies the factor base size in his article, i.e. the number of primes in the factor base. This system, on the other hand, takes as parameter the upper bound of the prime numbers constituting the factor base. For comparison, I wrote down both numbers in the tables below. Which one you use does not really matter, although there is no general one-to-one correspondence between the two. It depends on the number we are trying to factor,  $N$ , since the condition for membership in the factor base is that  $N$  is a quadratic residue of the prime (see section 2.1). This can vary from number to number, but is of the same order of magnitude for numbers of equal size.

The results of the test runs are as follows:

Size of the Number	Block Size	M	Large Prime Bound	System
60 digits	200,000	1,500,000	250,000,000	Linux cluster/ Red Hat

Table 26: Parameters of the First *Factor Base Size Comparison* Test Runs.

The suggested factor base size is 3000 for 60-digit numbers which equals a factor base prime bound of around 60,000.

Note also: The large prime bound is held constant here, instead of being deduced from the factor base prime bound. This means the smaller the factor base size, the more partial relations are found and vice versa.

<b>Factor Base Prime Bound</b>	<b>Factor Base Size</b>	<b>Running Time</b>	<b>Sieving Time</b>	<b>Trial Division Time</b>	<b>Number of Polynomials</b>
15,000	864	1856.62	1395.53	6.55	17414
20,000	1128	1114.33	808.93	9.08	9764
25,000	1382	816.27	573.27	13.15	6758
30,000	1650	638.02	432.35	16.63	4917
40,000	2161	443.77	286.1	23	3140
50,000	2612	384.1	229.88	32.33	2424
55,000	2839	354.6	202.2	37.28	2159
60,000	3076	338.85	184.02	41.47	1951
65,000	3293	330.12	172.37	47.07	1802
70,000	3520	313.87	162.73	51	1668
75,000	3750	312.32	155.58	56.98	1575
100,000	4849	294.87	123.97	81.15	1183
125,000	5884	313.42	108.03	113.1	975
150,000	6915	336.12	96.32	149.05	853
200,000	8966	406.52	88.28	224.75	704
250,000	10966	487.1	80.9	305.8	619
300,000	12992	588.13	80.23	403.9	560
350,000	14988	712.32	78.45	518.98	518
500,000	20836	1106.57	78.9	893.35	441
750,000	30138	1847.48	83.4	1590.53	386

Table 27: Results of the First *Factor Base Size Comparison* Test Runs.

Put into a diagram, it looks this way: The x-axis shows the factor base prime bound in logarithmic scale, the primary y-axis shows the curves for the running time (solid line), the sieving time (dotted line) and the trial division time (dotted-dashed line) and the secondary y-axis shows the curve for the number of polynomials generated (dashed line).

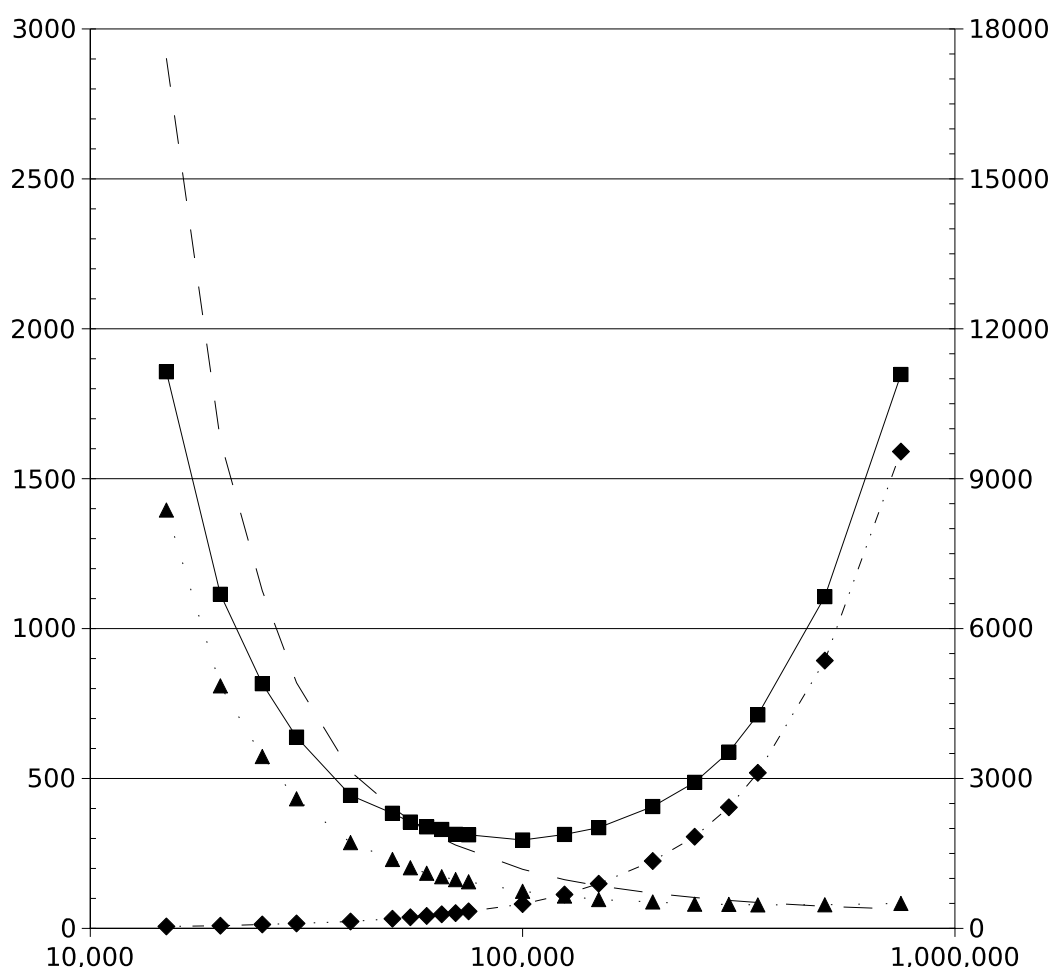


Figure 22: Diagram of the First *Factor Base Size Comparison* Test Runs.

One can see that the curves are very steep once we get out of the optimal range, so we want to avoid choosing the wrong value for the factor base prime bound. The cost is too high. Anyhow, the running time for the suggested value (60,000) is pretty low. The lowest value for this system is reached at ca. 100,000, in other words at a factor base size of nearly 5000. This is  $\frac{2}{3}$  higher than what Silverman took. The optimal range seems to be from 40,000 to 250,000 (with a factor base size from ca. 2000 to 11000), giving running times between 290 and 500 seconds.

The shape of the curves in general is that of a cup where the sieving time and the trial division time meet slightly right of the spot where the running time has its minimum. The sieving time curve flattens out as the factor base prime bound increases. We can see that the number of polynomials generated drops very fast. This is of course because we find more smooth relations per polynomial with a larger factor base.

As we already saw in the previous section, finding more probably smooth relations also comes with a price. Additionally, the more primes we have in the factor base, the more trial division work for one single probably smooth relation we have. That is why the trial division time and consequently the running time shoot into the sky for larger factor bases.

So, what about 66-digit numbers?

The results of the test runs are as follows:

Size of the Number	Block Size	M	Large Prime Bound	System
66 digits	200,000	2,000,000	1,000,000,000	Linux cluster/ Red Hat

Table 28: Parameters of the Second *Factor Base Size Comparison* Test Runs.

Factor Base Prime Bound	Factor Base Size	Running Time	Sieving Time	Trial Division Time	Number of Polynomials
40,000	2116	4116.53	2878.7	46.72	26467
50,000	2604	3118.43	2082.88	64.43	18303
60,000	3050	2552.52	1606.9	79.43	14160
70,000	3494	2130.7	1327.17	96.55	11438
80,000	3938	1937.62	1162.42	112.3	9625
90,000	4375	1809.32	1036.38	128.87	8345
93,000	4517	1768.75	994.6	137.35	8018
100,000	4804	1652.88	935.77	146.25	7417
110,000	5226	1632.43	889.27	173.35	6734
125,000	5896	1556.95	785.42	215.93	5799
150,000	7013	1472.35	666.62	284.67	4763
175,000	8093	1451.2	605.47	351.12	4068
200,000	9133	1446.02	550.17	422.85	3581
250,000	11169	1549.5	489.23	565.42	2976

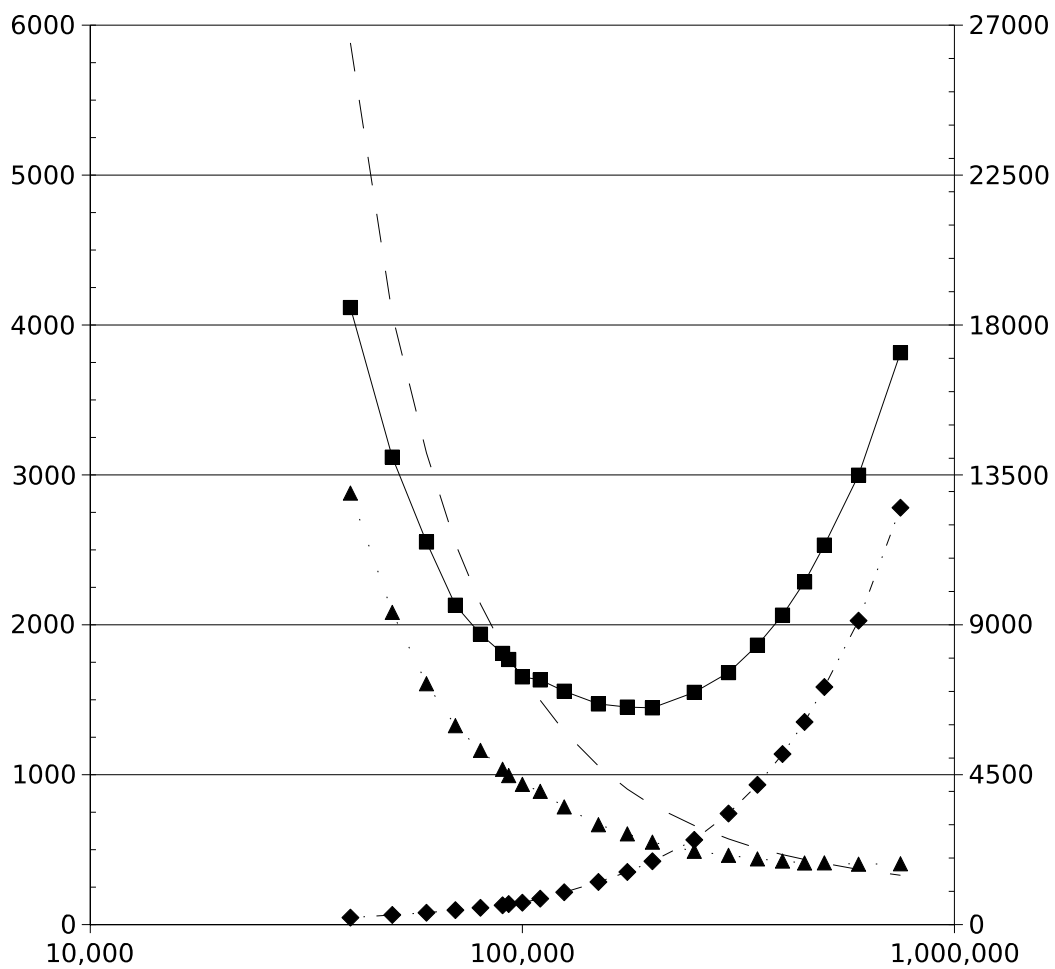
Table 29: Results of the Second *Factor Base Size Comparison* Test Runs.



Factor Base Prime Bound	Factor Base Size	Running Time	Sieving Time	Trial Division Time	Number of Polynomials
300,000	13155	1681.37	462.62	741.3	2574
350,000	15116	1864.27	438.6	932.62	2289
400,000	17044	2064.37	424.3	1137.92	2102
450,000	18966	2286.25	411.53	1351.78	1955
500,000	20868	2351.68	411.98	1585.13	1842
600,000	24654	2998.28	403.53	2027.8	1656
750,000	30217	3815.12	406.02	2781.08	1482

Table 29: Results of the Second *Factor Base Size Comparison* Test Runs.

The diagram is analogous to the one above.

Figure 23: Diagram of the Second *Factor Base Size Comparison* Test Runs.

The minimum in this curve can be found at about 200,000 which is twice as large as for 60-digit numbers. Silverman advised a factor base size of only 4500, whereas the optimal value observed here is just over 9000.

The optimal range is much narrower and goes from about 95,000 to 300,000 (with a factor base size from ca. 4600 to 14000), giving running times between 1450 and 1750 seconds (just below half an hour). There is much more space between the running time curve and the sieving and trial division time curves. This is of course because the sieving time curve and the trial division curve are steeper than before.

The numbers say that at the point of intersection of the sieving time curve and the trial division curve, which is very close to the optimum of the running time, they each consume around  $1/3$  of the total running time. With a total time of ca. 1500 seconds, the system spends about 500 seconds on sieving, 500 seconds on trial division and 500 seconds on other parts of the algorithm (mainly polynomial/factor base generation). Previously, it was stated that the sieving should take approximately  $2/3$  of the time (see section 5.3), but that was when the trial division time was constant and extremely low. Now, we see that the sieving time and the trial division time together should take about  $2/3$  of the time, which also fits in with the diagrams in section 5.3.

**Conclusion:** The factor base sizes proposed by Silverman are too small for this system. The optimal values for 60-digit numbers and 66-digit numbers are ca. 5000 respectively 9000. These values are reached with a factor base prime bound of about 100,000 resp. 200,000.

**Summary:** We can save much time by choosing optimal parameters for our system. Therefore, one should always do test runs before factoring a number with a new size. The table below shows some approximately optimal parameters for this system.

Size of the Number	Block Size	M	Large Prime Bound	Factor Base Prime Bound
60 digits	200,000	1,500,000	250,000,000	100,000
66 digits	200,000	1,500,000	?	200,000
70 digits	200,000	?	100,000,000	?

Table 30: Some Optimal Parameters.

## 5.6 Environment Comparison

This section compares the running times on the different systems presented in section 3.4. Additionally, it shows the results of test runs for the different compilers on the Linux cluster/Red Hat system.

During the first series of test runs, I discovered that the timings were faulty. This is a bug in LIP and therefore, all times given in this chapter are not in seconds, but rather in 0.6 seconds. That is to say, one would have to multiply every value (above and below) with 0.6 to get the correct numbers. However, this does not change the results in principle or influence any conclusions drawn.

### 5.6.1 System Comparison

The results of the test runs are as follows:

Size of the Number	Block Size	M	Large Prime Bound	Factor Base Size
60 digits	200,000	1,400,000	450,000	3035

Table 31: Parameters of the *System Comparison* Test Runs.

To be able to compare the expected computing power with the achieved running times, I calculated a “Mips-index” whose value is  $10000/\text{Mips}$ , Mips being the value mentioned in table 1 in section 3.4.

Operating System	Running Time	Sieving Time	Mips-Index
Linux cluster/Red Hat	355.4	210.13	2.28
Linux/Mandrake	307.67	183.05	2.08
Microsoft Windows	339.4	186.7	2.08
Linux/Fedora	248.12	146.55	1.58
Linux/Red Hat	465.42	307.15	3.27
Linux/Debian	570.92	376.07	4.19
Unix/Sun Solaris	1658.45	1076.23	11.56
IRIX	DNF	DNF	

Table 32: Results of the *System Comparison* Test Runs.

Put into a diagram, it looks this way: The x-axis shows the system, the primary y-axis shows the bars for the running times with the sieving times as dark grey bars and the secondary y-axis shows the curve for the Mips-index.

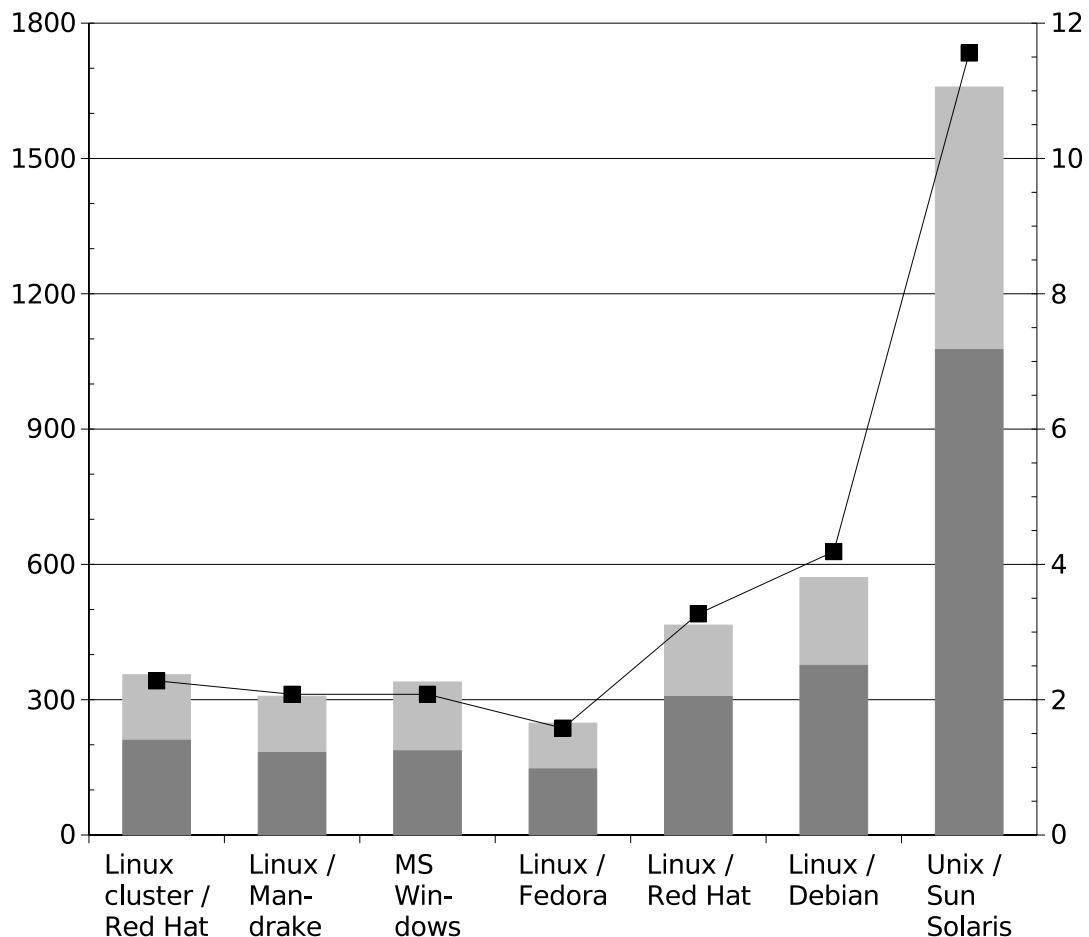


Figure 24: Diagram of the *System Comparison* Test Runs.

The result is: The Linux/Fedora system is clearly the fastest, followed by the Linux/Mandrake system and the Linux cluster/Red Hat system. They all have running times under 400. The Linux/Red Hat system and the Linux/Debian system are a bit slower, probably a little because of the used block size which does not quite let one whole block fit into the processor cache. The Unix/Sun Solaris system is the slowest and the IRIX system did not even finish within two hours.

The Microsoft Windows system is on the same computer as the Linux/Mandrake system and therefore the running times do not differ much. We can state that the algorithm runs a bit slower under Windows, since we have to go via Cygwin.

The Mips-index fits perfectly onto the running times.

What can be said about the Unix system and the IRIX system?

Is Linux better for this type of computation?

The answer is yes and no. It is not so much the operating system, but the computer that is running it. The environment as such may be suited for memory intense calculations like weather prediction or graphical rendering. For processor intense calculations like sieving however, we need a higher clock speed rather than a larger amount of memory. This applies to the size of the numbers for which test runs were done. If we were to factor much larger numbers, we would exceed the RAM of the Linux systems and then maybe the Unix system and the IRIX system would not be that slow in comparison. Also, with a parallel implementation of the critical phases of the algorithm, a multiprocessor environment like the IRIX system could be used more effectively. However, the idea behind this system was to distribute the labour and thus reducing the load put on each single sieve. The breakup into blocks assures that we can always scale down the amount of computation required by one sieving step. We only need more blocks for larger numbers. The server load on the other hand would increase heavily with much larger numbers.

### 5.6.2 Compiler Comparison

It is said that the Portland Group compiler and the Intel compiler usually give faster runs than the GNU compiler, because they exploit the computer architecture more effectively (see section 3.4). Is that true for this system?

Note: Due to differences between the compilers, some compilation flags had to be adjusted in some way. This makes the comparison less accurate, but hopefully as accurate as possible.

The tests were performed on the Linux cluster/Red Hat system.

The results of the test runs are as follows:

Compiler	Running Time	Sieving Time
GNU Compiler Collection (gcc)	355.4	210.13
Portland Group's Compilers (pgCC/pgcc)	680.92	509.33
Intel's Compiler (icc)	532.98	402.27

Table 33: Results of the *Compiler Comparison* Test Runs.

Put into a diagram, it looks this way: The x-axis shows the compiler and the y-axis shows the bars for the running times with the sieving times as dark grey bars.

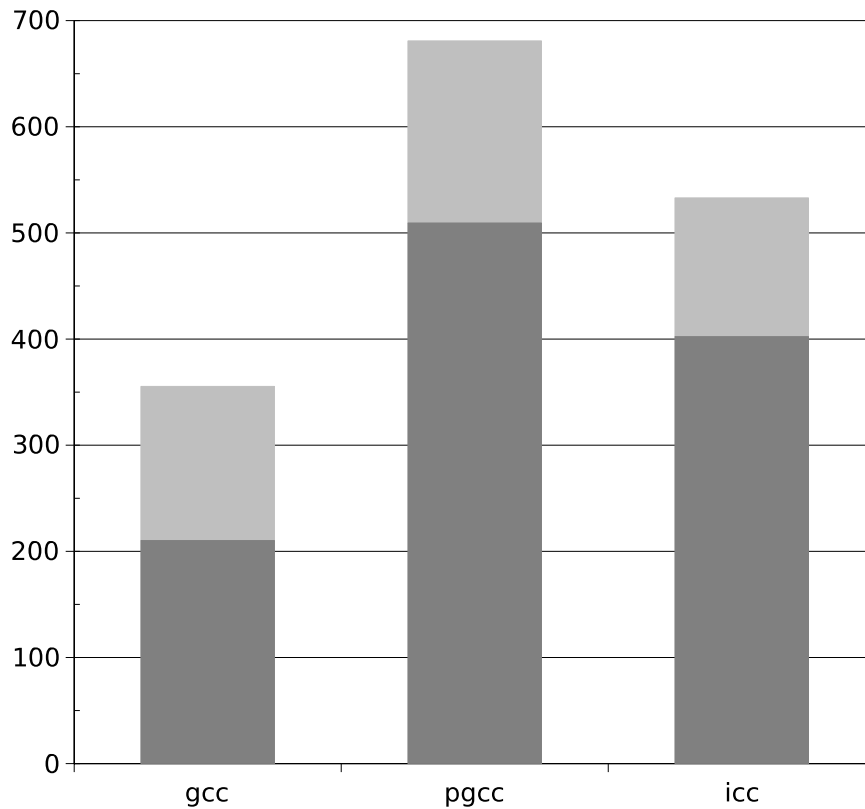


Figure 25: Diagram of the *Compiler Comparison* Test Runs.

We can see that the above statement does not concur with our observations. This may be because the flag that gives as much optimisation as possible with gcc was used. It can also be seen that the differences are substantial and thus it matters which compiler/compiler flags you use.

## 6 Conclusions

---

This chapter is divided into two parts. The first part is a summary of the results observed and discussed in detail in the previous chapter, and the second part deals with my personal experiences with this thesis work.

### 6.1 Results

A general conclusion drawn from the results of the previous chapter is that it is good practice to perform test runs before the system is taken into use. Parameter tuning should be applied to all new versions of the program and for the sizes of the numbers to be factored. However, it is not particularly useful to do test runs on different computers, since the Mips rate seems to be an adequate benchmark for the speed of the system already and there is no point in having separate parameter sets (see section 5.6).

About the parameters in detail:

- The size of the number is given implicitly. We have seen that the running time increases exponentially with growing number size. For an 80-digit number, factorisation takes about four hours on a modern computer.  
For more information, see section 5.1.
- The number of polynomials is not actually a parameter, but dependent on the other parameters. The system generates as many polynomials as necessary to find the required amount of relations.
- The block size should be chosen between 100,000 and 200,000 for processors with 256-512 KB cache. Rather choose a smaller block size so the entire block fits into the cache (see section 5.2).
- The sieving bound  $M$  should neither be taken extremely low nor extremely high (see section 5.3). The value seems to be just right when the sieving time and the trial division time together are about  $2/3$  of the total running time.
- The large prime bound suggested by Silverman was too high for this system. Optimal values seem to be 250,000,000 for 60-digit numbers and 100,000,000 for 70-digit numbers (see section 5.4).
- The size of the factor base on the other hand should be chosen bigger than in Silverman's article. This is a crucial parameter which has a high influence on the running time (see section 5.5).

## 6.2 Personal Experiences

I wish I had had several years to work on this project, complete it, refine, improve and extend it.

I spent the first few weeks on acquiring knowledge I never had the chance to apply, because my estimation of needed time and work was a little off. Also, I did not have a supervisor that was familiar with the subject which made it harder to get help with the theory and with planning the coverage of the thesis. I had planned more than I could manage in the time frame of a master's thesis. Later, in the implementation phase, I discovered I wanted to apply the block Lanczos algorithm and spent lots of time figuring *that* out. I know that a master's thesis should be independent work, but too much independent work from my side led to dead time when I struggled to understand things I had to ask about later anyway. Once again, thanks to all the people who were patient enough to answer all my questions. That really helped a lot!

Unfortunately, I did not even finish the standalone application. Here is the status: I still do not know exactly what to do when block Lanczos terminates with  $V_m = \mathbf{0}$ . I modified the block Lanczos implementation over and over towards the end, so now the system cannot factor any numbers at all. Something went wrong. At some point in the past, I could actually factor some smaller numbers (about 26 digits), but that was pure luck. I know the implementation of block Lanczos was faulty already then. However, I am positive that at least the sieving part is working the way it should (even if it runs out of memory for larger numbers).

I had thought the implementation work would have been simpler than it was. It is so easy to be fooled to believe that the transition from theory to practice is straightforward, as long as you are certain you understand the theory. The biggest problem for me was that the language C/C++ allows for so many nonsense constructs and obviously erroneous code. One should think it would be easy to discover such errors since they are so obvious, but it is not. For one thing, one can be blind for the most stupid errors (especially when fabricated by oneself) and for the other, it is often hard to find the source of runtime errors when the amount of code grows. Also, it was difficult for me to maintain a good structure throughout the code along the way. This is because I had to adjust my code so many times when I came to understand the working of the algorithms a little better.

All in all, it was fun to try and work this out on my own. I enjoyed the subject and I am interested in learning more in my spare time.



## 7 Future Work

---

There is very much to be done for this system. Not only due to the fact that the implementation is unfinished. This system is a potential lifetime project and my list of ideas for the system's future is a colourful mixture of things I had in mind for this thesis, optional extensions I would like to see implemented and things that could be considered useful.

The ideas below are written as a bullet list to make it more convenient to choose items to implement, tick off items and change, add or remove items. Additionally, I arranged the items by category.

The most basic things to be done are:

- Finish the standalone application.
- Finish the implementation of the server.
- Write a client that can communicate with the server.
- Implement other factorisation methods as proposed.

Other propositions for extensions are:

- Write an administration program for the server side that interfaces directly with the database.
- Automate the building process with makefile generation via GNU configure (see [26]).

Modifications and additions for the program itself:

- Exchange hard coded strings such as "params.txt" with something more flexible.
- Utilise environment parameters:
  - Have different block sizes for different computers depending on the size of the processor cache.
  - Add a means of measuring the Mips rate for estimation of the running time.
    - Deduced from that estimation, give clients a possibility to order a specific amount of computation.
- Employ parallel programming techniques for optimal use of multiprocessor systems.

Proposals applying more or less specifically to the quadratic sieve:

- Try to calculate certain parameters based on the number to be factored or collect some sets of standard parameters.
- Speed up computation by implementing time-critical arithmetic operations in the matrix elimination part and sieving part of the algorithm in assembler.
- Do not merge partial relations in memory. Instead, estimate the amount of partial relations beforehand and collect them in a file.
- Rewrite the classes in the *QSMatrix* files using templates with the size of the matrix/vector as parameter.
- Implement other sieving variants, e.g. with several large primes.
- Implement other matrix elimination methods.
- Implement other main variants of this method, for example SIQS.

Ideas for the client, server and network communication:

- Implement the mechanisms described in section 4.3.
- When requesting value ranges, have the option of subscribing to a certain project and let the client send the results and download new value ranges automatically until either the project is finished or the subscription is cancelled.
- Redistribute sieving intervals in case the server does not receive a result from a client within a certain time frame.
- Log all user actions and server responses.
- Generate statistical data on users and projects.
- On the server side, back up all incoming results in a separate directory for each user.
- Write a client GUI.
- Deploy smarter error handling, that is to say do not simply drop garbage messages. Also try to filter out messages that attempt to execute code.
- Implement error detection/notification on the server side.
- Develop error recovery for inconsistent data in the database and commands that were only partly executed. Provide automatic recovery from server errors.
- Establish password requirements, such as password length, use of digits, mixed case and blocking of dictionary words.
- Encrypt passwords.
- Use encryption for all messages with the help of SSL sockets.
- Add a hash code for transferred data to filter out faulty messages more easily and to avoid storing nonsense data at the server side.

## 8 References

---

### 8.1 Books

- 1 Riesel, Hans (1994) *Prime Numbers and Computer Methods for Factorization*. (2nd Ed.) Boston: Birkhäuser. ISBN 0-8176-3743-5
- 2 Bressoud, David M. (1989) *Factorization and Primality Testing*. New York: Springer-Verlag. ISBN 0-387-97040-1
- 3 A.K. Lenstra, H. W. Lenstra Jr. (1993) *The development of the number field sieve*. Berlin: Springer-Verlag. ISBN 3-540-57013-6

### 8.2 Internet

- 4 Eric Weisstein's world of Mathematics  
URL: <http://mathworld.wolfram.com/>
- 5 CiteSeer - Scientific Literature Digital Library  
URL: <http://citeseer.nj.nec.com/cs>
- 6 NFSNET - Large-scale Distributed Factoring  
URL: <http://www.nfsnet.org/>
- 7 The ECMNET Project  
URL: <http://www.loria.fr/~zimmerma/records/ecmnet.html>
- 8 ECM client/server  
URL: <http://www.interlog.com/~tcharron/ecm.html>
- 9 FermatSearch - DistributedSearch for Fermat Number Divisors  
<http://www.fermatsearch.org/>
- 10 The Cunningham Project  
URL: <http://www.cerias.purdue.edu/homes/ssw/cun/>
- 11 RSA Laboratories  
<http://www.rsasecurity.com/rsalabs/index.html>
- 12 Homepage of Arjen K. Lenstra (LIP source)  
<http://www.win.tue.nl/~klenstra/>
- 13 LiDIA - A C++ Library For Computational Number Theory  
<http://www.informatik.tu-darmstadt.de/TI/LiDIA/>
- 14 Cygwin Information and Installation  
URL: <http://www.cygwin.com/>

- 15 PostgreSQL  
URL: <http://www.postgresql.org/>
- 16 unixODBC  
URL: <http://www.unixodbc.org/>
- 17 The freeodbc++ project (libodbc++)  
URL: <http://libodbcxx.sourceforge.net/>
- 18 Sendmail Home Page  
URL: <http://www.sendmail.org/>
- 19 The Portland Group Compiler Technology  
URL: <http://www.pgroup.com/>
- 20 Intel Software Development Products  
URL: <http://www.intel.com/software/products/>
- 21 Valgrind  
URL: <http://valgrind.kde.org/>
- 22 Concurrent Versions System - The open standard for version control  
URL: <http://www.cvshome.org/>
- 23 nsieve  
URL: <ftp://ftp.nosc.mil/pub/aburto/>
- 24 The Prime Pages  
URL: <http://www.utm.edu/research/primes/>
- 25 FactorWorld!  
URL: <http://www.crypto-world.com/FactorWorld.html>
- 26 The GNU configure and build system  
URL: <http://www.airs.com/ian/configure/>

### 8.3 Publications

- 27 Contini, Scott Patrick (1997) *Factoring Integers with the Self-Initializing Quadratic Sieve*. Master's thesis at the University of Georgia, USA.  
URL: <http://citeseer.nj.nec.com/contini97factoring.html>

- 28 Montgomery, Peter L. (1995) *A Block Lanczos Algorithm for Finding Dependencies over  $GF(2)$* . In: Advances in Cryptology - EUROCRYPT '95 (Saint-Malo). Berlin: Springer-Verlag, pp. 106-120, 1995.
- 29 Silverman, Robert D. (1987) *The Multiple Polynomial Quadratic Sieve*. Mathematics of Computation 48 (1987), pp. 329-339.
- 30 A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard (1990) *The number field sieve*. In: Proceedings of the 22nd Annual ACM Symposium on the Theory of Computation, pp. 564-572, 1990.  
URL: <http://citeseer.nj.nec.com/lenstra90number.html>
- 31 Elkenbracht-Huizing, Marije (1997) *Factoring Integers with the Number Field Sieve*. Ph. D. thesis, Leiden University, Netherlands.  
URL: <http://ub.leidenuniv.nl/>



## A Early Factorisation Methods Appendix

---

This appendix gives an introduction to some of the first factorisation methods. It is neither complete nor a detailed description. The interested reader can learn more in [1], [2].

### A.1 Trial Division

Applying the *Sieve of Eratosthenes*, one can easily find the primes up to a certain number. If the number to be factored can be divided by one of those primes, we have found a factor. Since the factorisation into primes is unique apart from the order of the factors, this method is guaranteed to generate a factor sooner or later. In other words, this method is deterministic.

Furthermore, every composite number must have a factor greater than 1 and smaller or equal to the square root of the number. Otherwise, the number is prime. Therefore, only primes smaller or equal to the square root need to be considered.

Trial division is only feasible for small numbers and/or numbers having small factors.

### A.2 Fermat's Factorisation Method

The idea of Fermat's factorisation method is to rewrite the number to be factored as a difference of two squares:  $n = x^2 - y^2$ . This is the same as:  $n = (x + y) \cdot (x - y)$  which is a product of two integers and hence a factorisation of the number. The problem is reduced to finding  $x$  and  $y$ . Fermat proposed starting at the square root of  $n$  for the variable  $x$  and searching for a suitable  $y$ . If none can be found,  $x$  is increased in the next iteration of the algorithm.

This method works for all numbers  $n$  if we search long enough.

Clearly, this method is not made for large numbers either, because it can take many iterations before a correct  $x$  is found. However, it works quite well with small numbers having a factor near the square root of the number.

### A.3 Legendre's Congruence

Fermat's factorisation method is an application of Legendre's congruence. It states that  $x^2 \equiv y^2 \pmod{n}$  has integer solutions  $x, y$  with  $x \neq \pm y$  for composite numbers  $n$ . This means that  $x^2 - y^2 \equiv 0 \pmod{n}$  which is a more general form of the equation Fermat used.

## A.4 Gauss' Factorisation Method

Gauss' factorisation method is "the basis of many sieving methods for factorization". (see [1], page 152)

It makes use of quadratic residues in the following way:

Given is a composite integer  $N$  that we want to factor. Take  $x$  close to  $\lfloor \sqrt{kN} \rfloor$  for some integer  $k$ . Then,  $a \equiv x^2 \pmod{N}$  will produce small quadratic residues. These can be broken into prime factors and recombined producing new small quadratic residues.

Every prime factor  $p$  of  $N$  must have the same quadratic residues, that is to say,  $a \equiv x^2 \pmod{p}$  for  $a \equiv x^2 \pmod{N}$ . With the smallest quadratic residues  $a_i$ , you can now calculate the values of Legendre's symbol  $(a_i/p)$  and get expressions for  $p$  of the form  $p = \alpha_i k + \beta_i$  or  $p = \alpha_i k - \beta_i$ . Some of them can be merged to obtain one or several simpler expressions for  $p$ . All the primes satisfying the expressions and being less than  $\sqrt{N}$  can be listed. Cross-checking with other quadratic residues of  $N$  (and thus of  $p$ ), eventually only one  $p$  will remain as a possible factor.

The problem with this method is that you need quite a number of small quadratic residues for larger integers and therefore need to calculate much towards the end of the method.

## A.5 Legendre's Factorisation Method

This method is similar to Gauss' factorisation method. The difference lies in the initial production of small quadratic residues of  $N$ . Legendre uses the continued fraction expansion of  $\sqrt{N}$  as described in section C.2.



## B Modern Factorisation Methods *Appendix*

---

Apart from the factorisation methods described in chapter 2, appendix D and appendix E, there are other modern factorisation methods. They show some good approaches towards solving the difficult task of factorisation for larger numbers (see also [1] and [2]). All the algorithms presented here are probabilistic meaning that we search in a certain direction without being able to predict the outcome. There is no guarantee that a result is found in a finite amount of time.

### B.1 Pollard $p-1$

This method published by John M. Pollard bases on the idea that if  $p-1$  divides  $Q$  then  $p$  divides  $a^Q - 1$ , if  $\gcd(a, p) = 1$  (due to Fermat's theorem).  $p$  is the factor of  $n$  we search for and we can determine it by calculating  $\gcd(a^Q - 1, n)$ .

The question is how to find appropriate parameters  $a$  and  $Q$ . Choose any value for  $a$ , e. g.  $a = 11$ . Now, suppose,  $p-1$  is a smooth number. Then, we can generate  $Q$  by multiplying small primes systematically. As we go along, we check the growing  $Q$  periodically to see if we have found a factor. We set a boundary for the included primes so as not to exceed some fixed value  $k$  for the highest power involved, that is  $p_i^{\alpha_i} \leq k$ . If no factor is found, we can continue by multiplying  $Q$  with primes greater than  $k$  and smaller than some  $l$  to find those  $p-1$  having only one larger factor. The Pollard  $p-1$  method is good at finding medium-sized factors fairly quickly. But if we are unlucky, there is no prime factor  $p$  where  $p-1$  consists of a lot of small factors and we don't get any result.

Bressoud writes: "Finally, the Pollard  $p-1$  algorithm is one of the reasons for the restrictions on the primes  $p$  and  $q$  in the RSA public key crypto-system. If  $p-1$  or  $q-1$  have only small prime factors, then Pollard  $p-1$  will crack the code very quickly." (see [2], page 69)

### B.2 Pollard Rho

The second method by John M. Pollard is called Pollard rho. It uses an irreducible polynomial  $f(x)$  applied recursively in the following way:  $x_i \equiv f(x_{i-1}) \pmod n$ . Let  $p$  be a factor of  $n$  and define  $y_i \equiv x_i \pmod p$ . Soon, we will end up in a cycle where  $y_i = y_j$  for

some  $(i, j)$ , because there are at most  $p$  different values for  $y$ . The first few values are not part of the cycle and will not be repeated. They are *on the tail* of the Greek letter rho.

Taking two values on the cycle  $y_i = y_j$ , we get  $x_i \equiv x_j \pmod{p}$ . That is the same as  $x_i - x_j \equiv 0 \pmod{p}$ , i. e.  $x_i - x_j = k \cdot p$ . If  $x_i \neq x_j$ , we can calculate  $p$  (or a multiple of  $p$ ) as  $\gcd(n, x_i - x_j)$  and we are finished. The problem is that we do not have  $p$  to begin with and therefore cannot deduce  $y_i$  and  $y_j$ . The solution is to try different values for  $i$  and  $j$  until either finding suitable values or until we give up and try a different polynomial. Richard P. Brent suggested one should test the differences  $x_j - x_{2^k} \pmod{n}$  for  $3 \cdot 2^{k-1} < j \leq 2^{k+1}$ .

This method is like Pollard  $p-1$  good at finding medium-sized factors of  $n$ . If the smallest prime factor of  $n$  is too large, the algorithm will run for a very long time until finding it. We can save time, if we multiply successive differences and then calculate the  $\gcd$ .

### B.3 CFRAC

The continued fraction algorithm by Michael Morrison and John Brillhart is almost like the quadratic sieve algorithm described in section 2.1. The difference is that it builds on the continued fraction expansion of the square root of  $N$ . As explained in section C.2,  $A_{n-1}^2 \equiv (-1)^n Q_n \pmod{N}$ . The right side of the congruence substitutes the function  $f(r_i)$  of the quadratic sieve. We now try to factor the  $Q_n$ s over the factor base. If we find a product of different  $Q_n$ s which is a square by Gaussian elimination, we are back where we started and can apply the congruence above.

The advantage of this method over the quadratic sieve is that the  $Q_n$ s are potentially smaller than the  $f(r_i)$ s. The disadvantage is that the  $Q_n$ s can only be factored through trial division and most residues do not factor completely within the factor base.

Hans Riesel states that “Morrison and Brillhart’s method is one of the most efficient *general* factorization methods which has been put to extensive use on computers. It was the first method of subexponential running time [...]. During the 1970’s it was the principal method used to factor large integers, having no particular mathematical form to favor some other method.” (see [1], page 193)

## C Definitions

## Appendix

This chapter contains some definitions and theorems mainly taken from [1] and [2]. It is meant as a reference for those who lack the necessary knowledge to understand the chapters about the factorisation methods. For a more detailed discussion, see [1] or [2].

The theory of quadratic residues presented here can be applied to the quadratic sieve and the number field sieve (see chapter 2 and appendix D). The section about continued fractions can be applied to CFRAC (see section B.3), the one about elliptic curves can be applied to the elliptic curve method (see appendix E) and the one about number fields can be applied to the number field sieve (see appendix D).

### C.1 On Quadratic Residues

**Definition 2:** If  $a \equiv x^2 \pmod{n}$  and  $\gcd(a, n)=1$ , then  $a$  is called a **quadratic residue of  $n$** .

**Definition 3:** Take an odd prime  $p$  and an integer  $a$  relatively prime to  $p$  and define **Legendre's symbol**  $(a/p)$  as:

$\left(\frac{a}{p}\right) = +1$ , if  $a$  is a quadratic residue of  $p$  and  $\left(\frac{a}{p}\right) = -1$ , if  $a$  is a quadratic non-residue of  $p$ . If  $a$  is a multiple of  $p$ , let  $(a/p) = 0$ .

**Definition 4:** Take an odd integer  $n$ , which is either prime or composite, an integer  $a$  relatively prime to  $n$  and define **Jacobi's symbol**  $(a/n)$  as:

$$\left(\frac{a}{n}\right) = \prod_i \left(\frac{a}{p_i}\right)^{\alpha_i}, \text{ if } n = \prod_i p_i^{\alpha_i}.$$

Jacobi's symbol obeys the same laws as Legendre's symbol and the same theorems (as defined below) can be applied.

But it is not a generalisation of Legendre's symbol, since we can no longer be sure that  $(a/n) = -1$ , if  $a$  is a quadratic non-residue of  $n$ .

Quadratic residues play an important role in modern factorisation methods. Without proof and in short, here are some theorems for calculating with Legendre's symbols as stated in [1] and [2]:

**Theorem 1: Euler's criterion:**

$$\left(\frac{a}{p}\right) = a^{(p-1)/2} \bmod p .$$

**Theorem 2:**

$$\left(\frac{a}{p}\right) \cdot \left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right).$$

**Theorem 3: The quadratic reciprocity law:**

If  $p$  and  $q$  are odd primes, then

$$\left(\frac{p}{q}\right) = \left(\frac{q}{p}\right) \cdot (-1)^{\frac{1}{2}(p-1)\frac{1}{2}(q-1)} .$$

**Theorem 4:**

If  $a \equiv b \bmod p$ , then  $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$ .

**Theorem 5:**

If  $t^2 \equiv a \bmod p$ , and  $p = 4k - 1$ , then  $t \equiv \pm a^{(p+1)/4} \bmod p$  for a quadratic residue  $a$  of  $p$ .

**Theorem 6:**

The problem of finding  $\pm t$  such that  $t^2 \equiv a \bmod p$  for a quadratic residue  $a$  of  $p$  can be solved in polynomial time for the case  $p = 4k + 1$ . This can be achieved with the help of Lucas sequences (see [1]).

## C.2 On Continued Fractions

The contents of this section refers to Appendix 8 in [1].

**Definition 5: A continued fraction** is a fraction like this:

$$b_0 + \frac{a_1}{b_1 + \frac{a_2}{b_2 + \dots + \frac{a_n}{b_n}}}$$

and it is denoted in the following way:

$$b_0 + \frac{a_1}{b_1} + \frac{a_2}{b_2} + \dots + \frac{a_n}{b_n}$$

**Definition 6:** The numbers  $a_i$  are called **partial numerators** and the numbers  $b_i$  (apart from  $b_0$ ) are called **partial denominators**.

**Definition 7:** A continued fraction with all partial numerators equal to 1, all partial denominators being positive integers and  $b_0$  being an integer, is said to be **regular**.

**Definition 8:** The representation of a real number  $x$  by a regular continued fraction is called the **regular continued fraction expansion** of  $x$ .

For irrational numbers, this expansion is infinite. However, it can be truncated at any point and the resulting shorter expansion (called *convergent*) approximates the value of the complete expansion.

Continued fractions are of interest in number theory, because they can be used to find small quadratic residues of an integer  $N$ . For that, we need the regular continued fraction expansion of  $\sqrt{N}$ .

### C.2.1 Expansion of Square Roots

**Theorem 7:** The regular continued fraction expansion of the square root of a non-square integer  $N$  can be described as:

$$x_0 = \sqrt{N}, b_i = \lfloor x_i \rfloor, x_{i+1} = 1/(x_i - b_i), \text{ giving}$$

$$\sqrt{N} = b_0 + \frac{1}{b_1} + \frac{1}{b_2} + \dots + \frac{1}{b_n} + \dots$$

It can be proved that the expansion is always periodic.

**Theorem 8:** To make calculation of  $x_{i+1}$  easier and more accurate,  $x_i$  can be rewritten in the following way.

$$x_1 = \frac{1}{\sqrt{N} - P_1}, \text{ with } P_1 = \lfloor \sqrt{N} \rfloor \text{ following the definition above.}$$

This is the same as:

$$x_1 = \frac{\sqrt{N} + P_1}{(\sqrt{N} - P_1)(\sqrt{N} + P_1)} = \frac{\sqrt{N} + P_1}{N - P_1^2} = \frac{\sqrt{N} + P_1}{Q_1} = b_1 + \frac{\sqrt{N} - P_2}{Q_1}$$

$P_2$  shall assume the largest integer possible, where  $P_2^2 < N$ .

In similar fashion,  $x_i = \frac{Q_{i-1}}{\sqrt{N} - P_i} = \dots = b_i + \frac{\sqrt{N} - P_{i+1}}{Q_i}$ .

We have  $b_i = \lfloor x_i \rfloor = \left\lfloor \frac{\sqrt{N} + P_i}{Q_i} \right\rfloor$  and  $P_{i+1} = b_i Q_i - P_i$ .

### C.2.2 Application on Quadratic Residues

Theorem 9: For  $A_n/B_n$  being the  $n$ th convergent of  $\sqrt{N}$  with  $n \geq 0$ , the following holds:

$$A_{n-1}^2 - NB_{n-1}^2 = (-1)^n Q_n, \text{ with } Q_n \text{ as in theorem 8.}$$

If we reduce that *mod*  $N$ , we find that  $A_{n-1}^2 \equiv (-1)^n Q_n \pmod{N}$ , and hence  $(-1)^n Q_n$  must be a quadratic residue of  $N$ . It so happens that  $Q_n < 2\sqrt{N}$  and the continued fraction expansion of  $\sqrt{N}$  can provide small quadratic residues of  $N$ .

Calculating the continued fraction expansion is fast and easy.

But because of the periodicity of the expansion, it sometimes produces only a small number of quadratic residues. It can be overcome by expanding  $\sqrt{kN}$  instead, for a suitable  $k$ . This works, because each quadratic residue of  $kN$  must also be a quadratic residue of  $N$ .

### C.3 On Elliptic Curves

Definition 9: **Elliptic curves** are curves represented by a cubic equation of the form:  $y^2 = Ax^3 + Bx^2 + Cx + D$ .

We want to have rational coefficients and rational points on the curve. Take  $By^2 = x^3 + Ax^2 + x$  with  $B(A^2 - 4) \neq 0$ . These curves are symmetric with regard to the  $x$ -axis. Furthermore, if  $|A| > 2$ , the curve consists of two separate parts: a closed loop on one side of the  $y$ -axis and an open curve through origin on the other. It intersects the  $x$ -axis at both  $x_1 = 0$  and at  $x_{2,3} = (-A \pm \sqrt{A^2 - 4})/2$ . If

$|A| < 2$ , the loop disappears and the only intersection occurs at origin.

An important thing with elliptic curves is that a straight line through points  $Q_1$  and  $Q_2$  on the curve always intersects the curve at a third point  $Q_3$  (which may be a point at infinity).

**Definition 10: Addition** of points ( $Q_1 + Q_2 = -Q_3$ ) on an elliptic curve is carried out by drawing a straight line through  $Q_1$  and  $Q_2$  and noting the third point of intersection,  $-Q_3$  as result. If  $Q_1 = Q_2$ , a tangent to the curve is drawn in that point.

**Definition 11:** The **infinity point**  $I$  is the additive identity. It is also called zero point.

The following rules apply:  $I = -I$  and  $Q + I = Q$  for any  $Q$ .

**Definition 12: Multiplication** of a point  $Q$  on an elliptic curve with an integer  $n$  substitutes repeated addition, that is  $2Q = Q + Q$ ,  $3Q = Q + Q + Q$  and so forth.

**Definition 13: Homogeneous coordinates** can be used to keep things simple. Just replace the rational coordinates by fractions and all calculations can be carried out using integer coordinates.

Set  $x = \frac{X}{Z}$  and  $y = \frac{Y}{Z}$ . The new coordinates are  $(X, Y, Z)$ .

For the application of elliptic curves on the *elliptic curve method*, we need some arithmetic rules on the coordinate level. Since we only make use of the x-coordinate there, we will neglect the y-coordinate entirely. Of course, this is not really true, because it is part of every point, but we don't want to include it in any calculations.

**Theorem 10:** Doubling a point on an elliptic curve yields this result with  $2Q = 2(X_n, Y_n, Z_n) = (X_{2n}, Y_{2n}, Z_{2n})$  (see [1]):

$$\begin{cases} X_{2n} = (X_n^2 - Z_n^2)^2 \\ Z_{2n} = 4X_n Z_n (X_n^2 + AX_n Z_n + Z_n^2) \end{cases}$$

Theorem 11: To be able to calculate odd multiples of  $Q$ , let's have a look at  $(2n+1)Q = nQ + (n+1)Q = (X_{2n+1}, Y_{2n+1}, Z_{2n+1})$ .

From [1]:

$$\begin{cases} X_{2n+1} = Z_1((X_{n+1} - Z_{n+1})(X_n + Z_n) + (X_{n+1} + Z_{n+1})(X_n - Z_n))^2 \\ Z_{2n+1} = X_1((X_{n+1} - Z_{n+1})(X_n + Z_n) - (X_{n+1} + Z_{n+1})(X_n - Z_n))^2 \end{cases}$$

## C.4 On Number Fields

This section introduces definitions and theorems about number fields. The first part concentrates on quadratic fields, but most of the theory is applied on higher degree fields as well later on.

Definition 14: A **quadratic field**  $Q(\sqrt{D})$  consists of all numbers of the form  $z = r + s\sqrt{D}$  where  $r$  and  $s$  are rational numbers and  $D$  is an integer without any square factors.

Theorem 12: Every kind of field is closed under all four arithmetic operations, i.e. you can add, subtract, multiply and divide the elements and the result is an element of the field (with the exception of division by zero which is not allowed).

Definition 15:  $z$  is an **integer of the quadratic field** if it is a solution of  $z^2 + pz + q = 0$  with rational  $p$  and  $q$ .

Definition 16: The **conjugate** of a number  $x = r + s\sqrt{D}$  in  $Q(\sqrt{D})$  is  $\bar{x} = r - s\sqrt{D}$ .

Definition 17: The **norm** in  $Q(\sqrt{D})$  is denoted  $N(x) = N(\bar{x}) = x\bar{x}$ .

$$N(x) = \begin{cases} r^2 - Ds^2 & \text{if } D \equiv 2 \pmod{4} \text{ or } D \equiv 3 \pmod{4} \\ r^2 - rs - (D-1)/4 \cdot s^2 & \text{if } D \equiv 1 \pmod{4} \end{cases}$$



The following three definitions are valid both in quadratic fields and higher algebraic number fields without modification.

**Definition 18:** An integer  $x$  in  $Q(\sqrt{D})$  is called a **unit of the field** if  $N(x) = \pm 1$ .

**Definition 19:** Two integers in  $Q(\sqrt{D})$  are said to be **associated**, if their quotient is a unit.

**Definition 20:** An integer  $a$  in  $Q(\sqrt{D})$  is a **prime of the field**, if it cannot be written as the product of two integers in  $Q(\sqrt{D})$ , unless one of the factors is a unit. If  $a$  is not a prime of the field, it is said to be **composite in the field**.

That concludes the part about quadratic fields. The rest of the section deals with algebraic number fields of arbitrary degree  $n$ .

**Definition 21:** An **algebraic number** is one of the roots  $z$  of an irreducible polynomial with integer coefficients:

$$P(x) = a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n = 0 .$$

**Definition 22:** All the roots of  $P(x)$  are called the **conjugates of  $z$** .

**Definition 23:** The field generated by all rational expressions containing  $z$  is called an **algebraic number field  $Q(z)$** .

**Theorem 13:** Every number  $w$  in  $Q(z)$  can be written uniquely as  $w = c_0 + c_1z + c_2z^2 + \dots + c_{n-1}z^{n-1}$  with rational coefficients,  $z$  being one of the roots of  $P(x)$ .

**Definition 24:** An **algebraic integer in the number field** is a number  $w$  in  $Q(z)$  satisfying the following equation having integer coefficients:  $w^n + d_1w^{n-1} + \dots + d_{n-1}w + d_n = 0$ .

Definition 25: The **ring**  $\mathcal{O}$  consists of all algebraic integers in  $\mathcal{Q}(z)$ .

Definition 26: The **norm in the number field** is the product of all the conjugates of  $w$ .

$$N(w) = \prod_{i=1}^n w(z_i) \quad \text{with } w(z) = \sum_{i=0}^{n-1} c_i z^i$$

### C.4.1 Supplementary Theory

The following is collected from the second article in [3].

Theorem 14: The norm of  $a + bz$  is:  $N(a + bz) = a^d - t(-b)^d$ , with  $t = s \cdot r^{kd-e}$  and other parameters as defined in section D.1.

**Note:** In general:  $N(a + bz) = a^n - c_{n-1}a^{n-1}b + \dots + (-1)^n c_0 b^n$ , with  $P(x) = c_n x^n + \dots + c_0$ .

Definition 27: An integer of the number field is called **B-smooth** if its norm is B-smooth.

Definition 28: An **ideal**  $\mathfrak{a}$  of a ring is a subset that forms an additive group and has the property that if  $x$  belongs to the ring and  $y$  belongs to  $\mathfrak{a}$ , then the products  $xy$  and  $yx$  belong to  $\mathfrak{a}$ .

Definition 29: The **norm**  $\mathfrak{N}\mathfrak{a}$  of an ideal  $\mathfrak{a}$  in  $\mathcal{O}$  is the number of elements in  $\mathcal{O}$  divided by the number of elements in  $\mathfrak{a}$ .

Definition 30: A **first degree prime ideal**  $\mathfrak{p}$  is a non-zero ideal of  $\mathcal{O}$  whose norm is a prime  $p$ .

Theorem 15: Every  $\mathfrak{p}$  of  $\mathcal{O}$  can be mapped onto a pair of integers  $(p, c \bmod p)$  where  $p$  is the norm of  $\mathfrak{p}$  as before and  $P(c) \equiv 0 \bmod p$ . An integer  $w = \sum_i s_i z^i$  ( $s \in \mathcal{Q}$ ) of  $\mathcal{O}$  is in  $\mathfrak{p}$  iff  $\sum_i s_i c^i \equiv 0 \bmod p$ .

Theorem 16: If prime  $p$  divides the norm of  $a + bz$  in  $\mathcal{O}$  exactly  $k$  times, then a first degree prime ideal  $\mathfrak{p}$  with  $\mathfrak{N}\mathfrak{p} = p$  is a factor to the

$k$ th degree of the prime ideal factorisation of  $a + bz$ .

This is because if  $p$  divides  $a^d - t(-b)^d$ , then  $a \equiv -bc \pmod{p}$  for some  $c$  according to simple modular arithmetic. Therefore,  $a + bz$  is in  $\mathfrak{p}$  and  $\mathfrak{p}$  divides  $a + bz$ .

**Theorem 17:** A multiple of any generator  $\pi_{\mathfrak{p}}$  of prime ideal  $\mathfrak{p}$  is a prime factor of  $a + bz$  if  $\mathfrak{p}$  is in the prime ideal factorisation of  $a + bz$ .



## D The Number Field Sieve

## Appendix

This factorisation method was invented by John M. Pollard and makes use of number fields as the name suggests. Riesel states: “The number field sieve [...] is the most efficient method invented so far to factor numbers of the special form  $N = r^e + s$ , with  $r$  and  $s$  small integers and  $e$  possibly large.” (see [1], page 214)

### D.1 The Method

Like the quadratic sieve, the number field sieve searches for a congruence  $x^2 \equiv y^2 \pmod{N}$ .

**Definition 31:** A **ring homomorphism**  $\phi$  is a mapping from one ring to another that preserves the arithmetic operations addition and multiplication.

The number field sieve follows these steps factoring  $N$ :

- Find small integers  $r$  and  $s$  such that  $N = r^e + s$  for some  $e$ .

- Look for a suitable field  $Q(z)$ :

First, pick a degree  $d$  for the number field.

Then, calculate  $k$  such that  $(k-1) \cdot d < e \leq k \cdot d$ .

Define  $P(X) = X^d - s \cdot r^{kd-e}$ . With  $m = r^k$ ,  $P(m) \equiv 0 \pmod{N}$ .

Now, set  $z$  as a root of  $P(X)$ , that means  $P(z) = 0$ .

- Define the homomorphism  $\phi$  as:

$$\sum_{i=0}^{d-1} c_i z^i \rightarrow \sum_{i=0}^{d-1} c_i m^i \pmod{N}, \text{ i.e. every integer of the ring } Z(z) \text{ is}$$

mapped onto an ordinary integer  $y$  in  $Z$  with  $0 \leq y < N$ .

- Determine the factor bases  $FB1$  and  $FB2$  for factoring norms in  $Z(z)$  and integers in  $Z$ .
- Factor the norm of  $a + bz$  and the integer  $a + bm$  over  $FB1$ , respectively  $FB2$  in parallel for different  $a$  and  $b$ .
- In the same way as in the quadratic sieve, do elimination on a matrix to find  $a_i$  and  $b_i$  such that both  $P'(z)^2 \prod_i a_i + b_i z$  and  $P'(m)^2 \prod_i a_i + b_i m = y^2$  are squares.
- Calculate the value of the homomorphism:

$$\phi\left(P'(z)^2 \prod_i a_i + b_i z\right) = \sum_{i=0}^{d-1} c_i m^i \pmod{N} \equiv x^2 \pmod{N} \text{ which is a}$$

square, because the homomorphism preserves multiplication.

- We now have a solution to the congruence  $x^2 \equiv y^2 \pmod{N}$  and can check  $\gcd(x - y, N)$  for factors of  $N$ .

Just like in the quadratic sieve algorithm, we can allow for one large prime being outside the factor base. That increases our chances that the integers can be completely factored.

As mentioned, the strength of the algorithm is its efficiency for numbers of the form  $N = r^e + s$ . The difficulty lies in performing calculations in the number field. Whereas it is easy to factor small numbers in  $\mathbb{Z}$  for example, it is a lot more cumbersome in  $\mathbb{Z}(z)$ . Also, in some fields the prime factorisation may be ambiguous in  $\mathbb{Z}(z)$  which can lead to problems in finding a suitable square. These issues are discussed below.

## D.2 The General Number Field Sieve

The general number field sieve (GNFS) is a variation of the number field sieve that does not have any demands on  $N$ . Of course, we must still find a fitting number field.

- Choose integer  $n$  about  $\left(\frac{3}{2} \ln N / \ln \ln N\right)^{1/3}$ .
- Take integer  $m \approx N^{1/n}$ .
- Rewrite  $N = \sum_{i=0}^n c_i m^i$ , with  $0 \leq c_i \leq m - 1$ .
- Check, if the polynomial  $P(x) = \sum_{i=0}^n c_i x^i$  having the same coefficients  $c_i$  is irreducible. If not, change some coefficients.
- Like before, search for a root  $z$  of  $P(x)$ .

Note: Since the number field sieve cannot factor all numbers like the general number field sieve, it is also called special number field sieve (SNFS).

## D.3 Number Field Sieve Implementation

There are several different approaches of applying the theory of number fields and ideals to the number field sieve. One way is to construct the ring of algebraic integers  $\mathcal{O}$  and trying to factor the algebraic integers  $a + bz$  in  $\mathcal{O}$ . This is difficult, since one cannot prove that a ring in the number field is equal to  $\mathcal{O}$ . Luckily, there are approximations for  $\mathcal{O}$  which can be determined in polynomial time and they suffice for the purpose of factoring algebraic inte-

gers.

The approach intended in this thesis, on the other hand, was to avoid making calculations in the number field since the number field can get considerably large in the case of the general number field sieve.

Another way to implement the number field sieve is taking two polynomials instead of one polynomial, searching for two squares (say  $\gamma^2$  and  $\lambda^2$ ) in the two resulting number fields (one square each). Then, use the resulting congruence  $\phi(\gamma^2) \equiv \phi(\lambda^2) \pmod{N}$ . (See [31].)

### D.3.1 Factor Bases

This section explains how to choose the factor bases  $FB1$  and  $FB2$ .

- First, choose  $B_1$  and  $B_2$  which are the boundaries for the involved prime factors. As in the improved quadratic sieve, there may be one larger prime factor and we set  $B_4$  as bound for it. (See section D.3.4 for a suitable choice of parameters.)
- Next, construct a set  $P$  that contains all the prime numbers below  $B_2$  and a set  $R$  of pairs of numbers  $(p, c)$  where  $p$  ranges over the prime numbers below  $B_1$  and the  $c$ s are the integers below  $p$  which are zeros of polynomial  $P$  modulo  $p$ .

These sets constitute the factor bases  $FB2$  and  $FB1$ .

Note: The set  $R$  represents all prime ideals that could divide  $B_1$ -smooth numbers  $a + bz$  in  $Z(z)$ .

### D.3.2 Sieving

Some complications arise from the fact that the ring of integers in  $Q(z)$  not always turns out to be the ring of algebraic integers  $\mathcal{O}$  and thus  $Z(z)$  may not be a unique factorisation domain. This expresses itself in the circumstance that definition 30 is not valid in  $Z(z)$ , that is to say not every prime ideal of norm  $p$  is of first degree. There are several situations to be considered, but the bottom line is that an algebraic number need not be a square in  $Z(z)$ , just because its norm is a square in  $Z$ . The interested reader is referred to the article “*Factoring Integers with The Number Field Sieve*” in [3]. The description in this section follows the instructions in the latter.

Define the set  $Q$  as the first  $\lceil 3 \log N / \log 2 \rceil$  pairs of prime numbers  $q$  above  $B_1$  and integers  $s$  below  $q$  where  $P(s) \equiv 0 \pmod{q}$  and **not**  $P'(s) \equiv 0 \pmod{q}$ . (Similar but not identical to the definition of  $R$ .)

$$\text{Set } e_{p,c}(a+bz) = \begin{cases} k & \text{if } a \equiv -bc \pmod{p} \text{ and } p^k \mid N(a+bz), \text{ i.e.} \\ 0 & \text{otherwise} \end{cases}$$

the power, to which the prime ideal equivalent to the pair  $(p, c)$  divides  $a + bz$ .

The rational sieve does the following:

- The first thing is to set boundaries  $a_{\min}$  and  $a_{\max}$  for  $a$  and a boundary  $b_{\max}$  for  $b$ . Take  $a_{\max} = -a_{\min}$ .
- Let  $b$  range over  $[1, b_{\max}]$ . For every  $b$ :
  - Initialise an array containing  $2 \cdot a_{\max}$  elements which represent all  $a$  belonging to the interval  $[a_{\min}, a_{\max}]$  with *sieve locations*  $s_a = -\log(a + bm) + \log B_4$ .
  - For all primes  $p \in FB_2$ , check all  $a$  if  $a \equiv -bm \pmod{p}$ . In that case,  $p$  divides  $a + bm$  and  $s_a$  is replaced by  $s_a + \log p$ . Additionally, one can sieve with prime powers, too.
  - Report those  $a$  where  $s_a \geq 0$  after sieving. These  $a + bm$  are likely to be  $B_2$ -smooth except for one factor below  $B_4$ .

The algebraic sieve then continues:

- For the primes below  $B_1$ , check whether there is a pair  $(p, c)$  in  $R$  such that  $a \equiv -bc \pmod{p}$  and **not**  $b \equiv 0 \pmod{p}$ . This means that the corresponding  $\mathfrak{p}$  is in the prime ideal factorisation of  $a + bz$  (see section C.4.1). As before, replace  $s_a$  by  $s_a + \log p$ .
- If now  $s_a \geq 0$  after the first sieve and  $s_a \geq \log |N(a + bz)|$  after the second sieve, make sure that  $\gcd(a, b) = 1$ . Those pairs are good candidates in the search for squares and form the set  $T$ .
- Next, define a map  $e$  from a pair  $(a, b)$  in  $T$  to a binary row vector. The first element in the image of the map is determined by the sign of  $a + bm$  and is 0 if  $a + bm > 0$  and 1 if  $a + bm < 0$ . The following elements are the exponents modulo 2 of the primes  $p$  below  $B_4$  in  $a + bm$ , analogous to the quadratic sieve. Then, add the values of  $e_{p,c}(a + bz) \pmod{2}$  as defined above for all pairs  $(p, c)$  in the set  $R$ . Last, append the values  $l_{a,b,q,s}$  depending on the Legendres' symbols  $((a + bs)/q)$  for all  $(q, s)$  in  $Q$ . If  $((a + bs)/q) = 1$ , set  $l_{a,b,q,s} = 0$ , otherwise set  $l_{a,b,q,s} = 1$ .

The map  $e$  provides us with the matrix on which we can perform matrix elimination (see section D.1).



According to the mentioned article in [3], the probability that  $P'(z)^2 \prod_i a_i + b_i z$  is a square in  $Z(z)$  is very high if we can find enough (namely  $[3 \log N / \log 2]$ ) pairs  $(q, s)$  for which the value of the Legendre's symbol is  $+1$  (so-called quadratic characters).

We have now obtained a set of pairs  $(a, b)$  which are  $B_1$ - resp.  $B_4$ -smooth. (Note that in practice  $B_1 = B_2$ .) Pairs without any large prime factor are called *full relations*. The pairs where  $a + bm$  contains a large prime factor are termed *partial relations*.

### D.3.3 Calculating The Square Root

The coefficients of  $\gamma = P'(z)^2 \prod_i a_i + b_i z$  are potentially very large, if we try to compute the product in a conventional way. Therefore, we need to reduce the expression before applying the homomorphism:

- Find a prime  $q$  for which  $P(X) \bmod q$  is irreducible.
- Calculate  $\gamma \bmod q$ .
- Find  $\delta_0$  such that  $\delta_0^2 \gamma \equiv 1 \bmod \mathfrak{q}$ , where  $\mathfrak{q}$  is the ideal consisting of all elements in  $Z(z)$  with coefficients divisible by  $q$ . This can be done by an algorithm for taking square roots in finite fields.
- Apply a Newton iteration  $\delta_j \equiv \delta_{j-1}(3 - \delta_{j-1}^2 \gamma) / 2 \bmod \mathfrak{q}^{2^j}$  to find  $\delta_j$  such that  $\delta_j^2 \gamma \equiv 1 \bmod \mathfrak{q}$ .
- Stop as soon as the coefficients of  $\beta \equiv \delta_j \gamma \bmod \mathfrak{q}^{2^j}$  do not change for a few successive values of  $j$ .

The sought square root of  $\gamma$  is  $\beta$  (i.e.  $\beta^2 = \gamma$ ).

We have  $\varphi(\gamma) = \varphi(\beta^2) = \varphi(\beta)^2$ .

### D.3.4 Choice of Parameters

$B_4$  should be between  $B_2^{1.2}$  and  $B_2^{1.4}$ . This is to ensure that one can eliminate most of the large factors, at the same time that one has the benefit of partial relations.

$B_1$  and  $B_2$  in turn should be taken equal to

$$\exp\left(\left(\frac{1}{2} + o(1)\right)(d \log d + \sqrt{(d \log d)^2 + 2 \log(N^{1/d}) \log \log(N^{1/d})})\right),$$

The  $o(1)$  is for  $N \rightarrow \infty$ .

The same expression is evaluated for  $a_{max}$  and  $b_{max}$  too.

The optimal choice for  $d$  in the SNFS is  $d = \left( \frac{(3 + o(1)) \log N}{2 \log \log N} \right)^{1/3}$ .

Other parameters are chosen according to section 2.2.5.

## E The Elliptic Curve Method

## Appendix

The elliptic curve method is due to Hendrik W. Lenstra and as the name suggests, it is based on elliptic curves.

The method is fairly simple:

- Choose the parameters for the elliptic curve  $By^2 \equiv x^3 + Ax^2 + x$  with  $B(A^2 - 4) \neq 0$ .
- Set a rational starting point  $P_1 = (x_1, y_1) = (X_1, Y_1, Z_1)$ .
- Calculate prime multiples of  $P_1 \bmod N$  recursively:  
 $P_{i+1} \equiv p_i \cdot P_i \bmod N$ , where  $p_i$  is the  $i$ th prime with any smaller prime  $p$  inserted whenever some new power of  $p$  is passed.
- Check regularly if  $1 < \gcd(x_{i+1}, N) < N$ .
- Stop searching as soon as  $p_i > B_1$  for some  $B_1$ .

If no factor of  $N$  can be found, the elliptic curve method can be continued in the following way:

- Note the x-coordinate  $x_m$  of the last point of the previous part.
- Set  $B' = x_m^3 + Ax_m^2 + x_m$  and check that  $\gcd(B', N) = 1$ .
- Work with the curve  $B'y^2 = x^3 + Ax^2 + x$  and compute all the prime multiples  $q_i Q \bmod N$  where  $Q = (x_m, y_m)$  is the last point of the previous part and  $B_1 < q_i < B_2$  for some upper bound  $B_2$ .
- Again, check now and then if we have found a factor.

Peter L. Montgomery suggests the following choice of parameters:

$$\left\{ \begin{array}{l} x_1 = 2 \\ m = 3, 4, 5 \dots \\ k = (x_1^2 - m^2) / (x_1(m^2 - 1)) , \text{ which gives us } m \text{ separate curves.} \\ A = k + 1/k \\ B = A + 2 \end{array} \right.$$



## People Index

### B

Brent, Richard P. (1946-) ..... 4, 5  
 Brillhart, John (1930-) ..... 4  
 Buhler, Joe P. .... 4

### C

Carl Pomerance ..... 11  
 Cavallar, Stefania ..... 4  
 Contini, Scott Patrick ..... 49  
 Cunningham, Allan J.C.  
 (1842-1928) ..... 3, 4, 6

### D

Dodson, Bruce A. .... 4

### E

Eratosthenes of Cyrene  
 (276-194 B.C.) ..... 2, 4, 85  
 Euclid of Alexandria  
 (about 325-265 B.C.) ..... 2, 4  
 Euler, Leonhard (1707-1783) ..... 2, 4

### F

Fermat, Pierre de  
 (1601-1665) ..... 2, 3, 4, 11, 85

### G

Galois, Evariste (1811-1832) ..... 4  
 Gauss, Carl Friedrich  
 (1777-1855) ..... 2, 4, 86

### K

Kraitchik, Maurice  
 (1882-1957) ..... 2, 4, 11

### L

Legendre, Adrien-Marie  
 (1752-1833) ..... 2, 4, 85, 86  
 Lehmer, Derrick Henry  
 (1905-1991) ..... 2, 4  
 Lenstra, Arjen K. .... 4, 19  
 Lenstra, Hendrik W. Jr. .... 4, 105  
 Leyland, Paul C. .... 4, 6

Lucas, Edouard F. (1842-1891) ..... 2, 4

### M

Manasse, Mark S. .... 4  
 Mersenne, Marin  
 (1588-1648) ..... 2, 3, 4  
 Montgomery, Peter L.  
 ..... 4, 12, 16, 19, 105  
 Morrison, Michael A. .... 4

### P

Pollard, John M. .... 2, 4, 99  
 Pomerance, Carl ..... 4, 11  
 Powers, R. E. .... 4  
 Pythagoras of Samos  
 (about 569-475 B.C.) ..... 4

### R

Riesel, Hans (1929-) ..... 4, 99

### S

Silverman, Robert D. .... 4, 16,  
 ..... 37, 38, 43, 60, 66, 67, 69, 72, 77

### W

Wagstaff, Samuel S. .... 4, 5, 6  
 Woodall, H. J. .... 3, 6



# Main Index

---

## A

addition of points ..... 93  
administration program ..... 79  
algebraic integer ..... 95, 100, 101  
algebraic number ..... 9, 95, 101  
algebraic number field ..... 20, 95  
algebraic sieve ..... 102  
algorithm ..... 1, 2, 9, 78, 85  
arithmetic operations ..... 80  
array ..... 13, 17, 102  
assembler ..... 80  
associated ..... 95

## B

base ..... 3  
basis ..... 16  
benchmark ..... 77  
binary ..... 21  
block ..... 17, 30, 47, 49, 51, 52,  
..... 56, 57, 60, 62, 74, 75, 77  
block Lanczos ..... 16, 78  
block Lanczos method ..... 13  
block size ..... 39, 41, 47, 48,  
..... 49, 51, 74, 77, 79  
blocksize ..... 50  
building process ..... 79

## C

C++ ..... 25  
C/C++ ..... 19, 78  
cache ..... 49, 50, 51, 74, 77  
CFRAC ..... 4  
client ..... 5, 6, 17, 21, 25, 27, 28,  
..... 30, 31, 32, 33, 34, 79, 80  
code ..... 25, 32, 34  
communicate ..... 79  
communication ..... 25, 30, 33  
communication control ..... 30  
compiler ..... 21, 73, 75, 76  
composite ..... 3, 11, 89, 95  
composite number ..... 85, 86  
computation ..... 79  
Concurrent Versions System ..... 21  
configure ..... 79  
congruence ..... 10, 25, 26, 100  
congruent ..... 9

conjugate ..... 94, 95, 96  
continued fraction ..... 7, 8, 90  
continued fraction expansion .....  
..... 86, 91, 92  
convergent ..... 91, 92  
creation ..... 30  
cryptography ..... 1  
Cunningham number ..... 3  
Cunningham project ..... 3, 5, 6  
Cunningham tables ..... 6  
CVS ..... 21  
Cygwin ..... 21, 74

## D

data structure ..... 29, 30  
database ..... 8, 20, 27, 28,  
..... 29, 34, 79, 80  
database management system ..... 20  
DBMS ..... 20  
Debian ..... 21, 22, 23, 73, 74  
debugger ..... 21  
debugging ..... 28  
deletion ..... 30  
design ..... 25, 29  
deterministic ..... 2, 8, 9, 85  
deterministic algorithm ..... 1, 8, 11  
diagram ..... 30, 33, 38, 39, 41, 42, 44,  
..... 46, 48, 49, 50, 51, 53, 57,  
..... 58, 59, 61, 62, 63, 64, 65,  
..... 66, 68, 69, 71, 74, 76  
dictionary words ..... 80  
digit ..... 29  
digits ..... 5  
distributed ..... 25, 27, 29  
distributed system ..... 1, 5  
divisor ..... 2, 3

## E

ECM ..... 1, 4, 5, 8  
ECMNET ..... 5, 6  
Elements ..... 4  
elliptic curve ..... 7, 8, 19, 92, 93, 105  
elliptic curve method ..... 1, 5, 6, 7,  
..... 8, 93, 105  
email ..... 29, 31, 32, 33  
encryption ..... 80  
entity ..... 8

entity-relationship diagram ..... 8  
 environment ..... 7, 21, 75  
 ER-diagram ..... 8, 29  
 error ..... 32, 33, 34  
 error detection ..... 80  
 error handling ..... 80  
 error message ..... 30  
 error recovery ..... 80  
 estimation ..... 79  
 Euler's criterion ..... 90  
 Euler's method ..... 4  
 exclusive-or ..... 15  
 expansion ..... 91, 92  
 exponent ..... 12, 26, 102

## F

factor ..... 5, 6, 10, 11, 12,  
 ..... 85, 94, 95, 100  
 factor base ..... 5, 8, 11, 13, 16, 25,  
 ..... 30, 53, 54, 55, 67, 69,  
 ..... 70, 72, 99, 100, 101  
 factor base prime bound ..... 60, 67,  
 ..... 68, 69, 70  
 factor base size ..... 67, 68, 69,  
 ..... 70, 71, 72, 77  
 factorisation ..... 1, 3, 5, 6, 7, 8, 11,  
 ..... 12, 20, 23, 25, 29,  
 ..... 31, 32, 77, 85, 101  
 factorisation algorithm .....  
 ..... 2, 9, 10, 19  
 factorisation method ..... 2, 5, 7, 8, 9,  
 ..... 79, 85, 86, 89, 99  
 Fedora ..... 21, 22, 23,  
 ..... 45, 46, 73, 74  
 Fermat number ..... 3, 8  
 Fermat numbers ..... 6  
 Fermat's factorisation method ..... 85  
 Fermat's method ..... 4  
 FermatSearch ..... 6  
 finite field ..... 103  
 first degree prime ideal ..... 96  
 flow chart diagram ..... 33, 35, 36  
 fraction ..... 8, 93  
 full relation ..... 25, 103  
 future work ..... 7

## G

Gauss' factorisation method ..... 86  
 Gauss' method ..... 4  
 gcc ..... 21, 75, 76  
 gcd ..... 9, 10

gdb ..... 21  
 general number field sieve .....  
 ..... 9, 100, 101  
 generator ..... 97  
 GNFS ..... 9, 100  
 GNU ..... 79  
 GNU compiler ..... 75  
 greatest common divisor ..... 2, 9  
 GUI ..... 80

## H

hard coded ..... 79  
 hash code ..... 80  
 homogeneous coordinates ..... 93  
 homomorphism ..... 99, 103

## I

icc ..... 21  
 ideal ..... 20, 96, 103  
 identity matrix ..... 15  
 image ..... 102  
 implementation ..... 7, 19, 20, 25, 79  
 infinite ..... 91  
 infinity ..... 93  
 infinity point ..... 93  
 input ..... 31  
 instruction ..... 30, 31  
 integer ..... 2, 9, 10, 13, 19, 89, 93,  
 ..... 94, 95, 96, 99, 100, 101  
 Intel compiler ..... 75  
 Intel's compiler ..... 21  
 IRIX ..... 22, 73, 74, 75  
 IRIX64 ..... 21  
 irrational number ..... 91  
 iteration ..... 14, 25

## J

Jacobi's symbol ..... 89  
 Java database connectivity ..... 20  
 JDBC ..... 20

## K

key size ..... 3

## L

Lanczos method ..... 13, 14, 16  
 large factor ..... 25  
 large prime ..... 12, 13, 46, 62,  
 ..... 100, 101, 103



large prime bound ..... 13, 60, 61, 62,  
 ..... 63, 64, 65, 66, 67, 77  
 large primes ..... 80  
 Legendre's congruence ..... 85  
 Legendre's factorisation method .....  
 ..... 86  
 Legendre's symbol ..... 9, 11, 86,  
 ..... 89, 102, 103  
 libodbc++ ..... 20  
 library ..... 19, 20  
 LiDIA ..... 19  
 linearly dependent ..... 14  
 Linux ..... 20, 21, 22, 23, 37,  
 ..... 45, 46, 50, 73, 74, 75  
 Linux cluster ..... 22, 23, 37, 46, 58,  
 ..... 60, 63, 65, 67, 73, 74, 75  
 LIP ..... 19, 26, 73  
 logarithmic ..... 57  
 logarithmic scale ..... 38, 44, 61, 68  
 Lucas sequence ..... 3, 90  
 Lucas-Lehmer primality test ..... 2

## M

M ..... 39, 40, 42, 52, 53,  
 ..... 56, 57, 58, 59, 77  
 makefile ..... 79  
 Mandrake ..... 20, 21, 23, 37, 50, 73, 74  
 map ..... 102  
 matrix ..... 12, 13, 15, 25, 80, 102  
 matrix elimination ..... 12, 13, 25,  
 ..... 55, 80, 99, 102  
 memory ..... 13, 22, 26, 46, 49, 75  
 merged relation ..... 25  
 Mersenne number ..... 3  
 Mersenne prime ..... 3  
 message code ..... 30  
 message text ..... 30  
 method ..... 10, 11, 20  
 Microsoft Windows ..... 21, 73, 74  
 Mips ..... 23, 77  
 Mips rate ..... 79  
 Mips-index ..... 73, 74  
 MIPSpro C/C++ compilers ..... 21  
 mod ..... 9, 10  
 modification ..... 30  
 modular arithmetic ..... 97  
 modulo ..... 15, 20, 101, 102  
 Montgomery modular arithmetic .....  
 ..... 19  
 MPQS ..... 9, 12, 16  
 multiple polynomial quadratic sieve

..... 9, 12  
 multiplication of a point ..... 93  
 multiprocessor environment ..... 75  
 multiprocessor system ..... 79

## N

network ..... 27  
 network communication ..... 80  
 network protocol ..... 25, 30  
 Newton iteration ..... 103  
 NFS ..... 1, 4, 9  
 NFSNET ..... 5, 6  
 norm ..... 94, 96, 99, 101  
 nsieve benchmark ..... 23  
 null space ..... 16  
 number field ..... 7, 9, 19, 94, 95,  
 ..... 99, 100, 101  
 number field sieve ..... 1, 5, 7, 8, 9,  
 ..... 10, 20, 99, 100, 101  
 number size ..... 37, 38, 39, 40, 41,  
 ..... 42, 43, 44, 45, 46, 77  
 number theory ..... 19, 91

## O

object structure ..... 25  
 ODBC ..... 20, 28  
 open database connectivity ..... 20  
 operating system ..... 21, 23, 28, 73, 75  
 orthogonal ..... 14

## P

parallel programming ..... 79  
 parameter ..... 2, 7, 9, 16, 25, 26, 27,  
 ..... 28, 30, 37, 39, 44, 47,  
 ..... 50, 56, 58, 67, 72, 77,  
 ..... 79, 96, 101, 103, 104, 105  
 partial denominator ..... 91  
 partial numerator ..... 91  
 partial relation ..... 25, 46, 60, 67, 103  
 partial relations ..... 80  
 password ..... 20, 29, 31, 32, 33  
 password requirements ..... 80  
 PC ..... 21, 22  
 perfect number ..... 3  
 periodic ..... 91  
 periodicity ..... 92  
 pgCC ..... 21  
 pgcc ..... 21  
 pie chart ..... 55  
 pie chart diagram ..... 54

pivot element ..... 15  
 plain text ..... 33  
 point ..... 8, 92, 105  
 Pollard p-1 ..... 4, 87  
 Pollard Rho ..... 4, 87  
 polynomial ..... 5, 9, 12, 13, 16, 20,  
     ..... 28, 30, 47, 48, 49, 52,  
     ..... 53, 54, 55, 56, 57, 60,  
     ..... 61, 62, 63, 65, 68, 69,  
     ..... 70, 72, 77, 95, 100, 101  
 Portland Group compiler ..... 75  
 Portland Group's (PGI) compilers ....  
     ..... 21  
 PostgreSQL ..... 20  
 power ..... 3, 6  
 primality test ..... 2  
 primality testing ..... 11, 19  
 primary y-axis ..... 38, 39, 42, 44, 48,  
     ..... 50, 53, 57, 61, 68, 74  
 prime ..... 3, 89, 90, 95  
 prime bound ..... 13, 16  
 prime factor ..... 1, 6, 10, 11, 26, 86, 97  
 prime factorisation ..... 100  
 prime ideal ..... 97, 101, 102  
 prime ideal factorisation ..... 97, 102  
 prime number ..... 1, 2, 8, 9, 10, 11,  
     ..... 12, 13, 19, 101, 105  
 prime table ..... 2  
 probabilistic ..... 2, 9  
 probabilistic algorithm ..... 9  
 probabilistic method ..... 1  
 processor ..... 49, 51, 77  
 processor cache ..... 49, 79  
 product ..... 85  
 programming language ..... 19  
 project ..... 3, 5, 6, 27, 29, 30,  
     ..... 31, 32, 33, 34, 80  
 proof ..... 2, 89  
 protocol ..... 30  
 public key cryptography ..... 1

## Q

QS ..... 4, 10, 11  
 quadratic character ..... 103  
 quadratic field ..... 94, 95  
 quadratic non-residue ..... 89  
 quadratic reciprocity law ..... 90  
 quadratic residue ..... 7, 9, 10, 11, 67,  
     ..... 86, 89, 90, 91, 92  
 quadratic sieve ..... 7, 8, 9, 10, 11,  
     ..... 12, 16, 25, 28, 80,

..... 99, 100, 101, 102

## R

RAM ..... 49, 75  
 rational sieve ..... 102  
 record ..... 6  
 Red Hat ..... 22, 23, 37, 46, 50, 58,  
     ..... 60, 63, 65, 67, 73, 74, 75  
 regular ..... 91  
 regular continued fraction ..... 91  
 relation ..... 25, 30, 60, 62, 69, 70, 77  
 relationship ..... 8, 30  
 reliability ..... 30  
 result ..... 30, 31, 33, 34, 80  
 result message ..... 30  
 ring ..... 96, 100, 101  
 ring homomorphism ..... 99  
 RSA ..... 1, 3, 4, 10  
 RSA algorithm ..... 3, 10  
 RSA challenge number ..... 3  
 RSA laboratories ..... 3  
 Running Time ..... 52  
 running time ..... 14, 16, 37, 38, 39, 40,  
     ..... 42, 44, 45, 47, 48, 49,  
     ..... 50, 51, 53, 54, 55, 56,  
     ..... 57, 58, 59, 60, 61, 62,  
     ..... 63, 64, 65, 68, 69, 70,  
     ..... 72, 73, 74, 75, 76, 77, 79

## S

search space ..... 1  
 secondary y-axis ..... 38, 39, 42, 44,  
     ..... 48, 50, 53, 57, 61, 68, 74  
 security ..... 3, 30  
 security algorithm ..... 1, 10  
 semi-smooth ..... 57, 60  
 sendmail ..... 20  
 server ..... 1, 5, 6, 20, 21, 25,  
     ..... 27, 28, 29, 30, 31,  
     ..... 32, 33, 34, 75, 79, 80  
 SGI Origin ..... 22  
 sieve ..... 5, 25, 59, 62, 102  
 sieve location ..... 13, 49, 102  
 Sieve of Eratosthenes ..... 2, 4, 10,  
     ..... 11, 23, 85  
 sieving ..... 11, 13, 17, 25, 26, 27, 28,  
     ..... 30, 32, 47, 49, 52, 53, 55,  
     ..... 60, 67, 72, 75, 78, 101, 102  
 sieving bound ..... 38, 39, 42, 43,  
     ..... 44, 52, 53, 54, 55,  
     ..... 56, 57, 58, 59, 77

<b>T</b>	
TCP/IP .....	30
templates .....	80
test run .....	7, 19, 21, 37, 38, 39, 40,
.....	42, 43, 44, 45, 46, 47, 50,
.....	51, 52, 54, 55, 56, 58, 60,
.....	63, 65, 67, 70, 72, 73, 75, 77
theorem .....	2, 94
theoretical background .....	7
time line .....	4
time variables .....	28
time-critical .....	80
transfer .....	30, 31, 32
trial division .....	4, 13, 55, 70, 85
trial division time .....	60, 61, 62,
.....	63, 64, 65, 68, 69, 70, 72, 77

## UML ..... 10

VW

# X

## Y

**Z**



## På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ick-ekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

## In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>